교육성교육정보쎈터

주체97(2008)년

차 례

머 리 말

현시대는 과학과 기술이 비약적으로 발전하는 정보산업의 시대, 콤퓨터시대이다.

콤퓨터의 출현과 정보기술의 비약적인 발전은 과학기술부문만이 아니라 사람들의 사회생활령역에도 큰 영향을 주고있다. 이리하여 지난 시기에는 환상적으로만 생각하여 오던 문제들이 현실로 되고있으며 자연을 정복하고 사회를 개조해나가는 사람들의 창조적능력과 힘은 더욱더 커지고있다.

위대한 령도자 **김정일**동지께서는 다음과 같이 지적하시였다.

《프로그람을 개발하는데서 기본은 우리 식의 프로그람을 개발하는것입니다. 우리는 우리 식의 프로그람을 개발하는 방향으로 나가야 합니다.》(《**김정일**선집》 제 15 권, 196 페지)

나라의 정보산업을 발전시키는데서 무엇보다 중요한것은 우리 식의 프로그람을 개발하고 그에 기초하여 여러가지 실용적의의가 있는 프로그람들을 많이 만들어냄으로써 인민경제의 과학화, 정보화를 다그쳐나가는것이다.

최근 세계적으로 정보산업이 급속히 발전하는데 따라 정보화사회를 건설하고 국가기관들의 사무행정사업과 경영업무활동을 콤퓨터화하려는 움직임이 높아지는데 맞게 적지 않은 나라들에서 자기 식의 조작체계를 개발하는데 많은 관심을 돌리고있다. 그중에서도 1991년에 개발된 Linux조작체계를 리용하여 프로그람을 개발하는 비중이 높아지고있다.

물론 아직도 많은 면에서 부족점들이 있어 사용자들에게 불편을 주고있지만 Linux조작체계는 그것이 가지고있는 고유한 특징 즉 원천공개형조작체계, 강력한 콤퓨터망기능, 다중가동환경, 강력한 보안, 다중과제-다중사용자의 지원 등으로 하여 최근에 널리 리용되고있는 조작체계이다.

이러한 특징으로 하여 Linux조작체계는 콤퓨터망봉사기의 조작체계로서 널리 리용되고있으며 매몰형조작체계로서도 많이 실용화되고 있다.

이러한 발전동향은 오늘 콤퓨터망전문가들과 관리자들이 Linux조작체계에 대한 보다 넓고 깊은 지식을 소유할것을 요구하고있으며 여기에 이목을 집중시키게 하고있다.

이 책에서는 이러한 요구를 반영하여 Linux조작체계를 리용한 콤퓨터망프로그람작성방법에 대하여 취급하고있다.

책에서는 콤퓨터망프로그람작성에서 기초로 되는 프로세스, 신호, 스레드에 대하여 설명하였으며 프로세스들사이 통신, 체계간 통신을 실현하는 방법들에 대하여 실례를 들어

설명하였다.

이 책은 Linux조작체계를 다루어보았고 C언어에 대한 초보적인 지식을 가진 사람들을 대상으로 하여 Linux가동환경에서 요구하는 체계프로그람을 개발하는데 도움을 주는 방향에서 집필하였다. 책에서 C언어만을 사용한것은 아니지만 C언어를 위주로 사용한것만큼 C언어의 문법에 대한 학습도 함께 하여야 이 책에서 설명하는 내용에 대하여 더 잘 리해할수 있다.

책의 내용과 서술에서 미숙한 점이 적지않으리라 보면서 Linux를 리용하여 콤퓨터망의 세계를 개척해 보려는 독자들에게 어느정도 도움이 되기를 기대한다.

제 1 장

콤퓨터망의 간단한 소개

서론

이 장에서는 리론적인 내용을 바탕으로 콤퓨터망(Network)전반에 대하여 소개하고있다. 처음 읽어보는 독자들에게는 리해하기 좀 어려운 부분이지만 망프로그람을 작성하려면 반드시 알아야 할 내용들이다. 이제 소개하려는 내용은 뒤에서 나오는 프로그람작성법에 대한 내용들과는 특별한 련관이 없지만 책의 첫머리에서 이러한 내용을 소개하는 리유는 콤퓨터망프로그람작성법을 다루면서 콤퓨터망에 대한 깊은 지식이 없이 프로그람을 작성한다는것은 기초가 없는 집을 짓는것과 같다고 말할수 있기때문이다. 그리고 프로그람작성에서는 코드의 내용보다도 기초리론과 원리가 더 중요하기때문이다.

여기서는 콤퓨터망에 대한 일반적개념과 규약을 간단히 소개하고 콤퓨터망에서 UNIX와 관련한 부분을 설명하였다. 이 장의 구성은 다음과 같다.

목표

1. 콤퓨터망의 일반적리해
2. 규약
3. UNIX콤퓨터망

제 1 절. 콤퓨터망의 일반적리해

과거의 고전적인 콤퓨터망개념과는 달리 최근의 콤퓨터망은 지난 시기와는 비교할수 없을 정도로 고속화되고있으며 복잡해지고있다. 많은 봉사들이 서로 통합되고 보다 완성되여 현대의 콤퓨터망은 큰 대역폭과 높은 성능을 요구하게 되였다. 그 결과 콤퓨터망상에서 문제로 되고있는 통신량의 과잉이나 시간지연을 줄이기 위한 대책들과 새로운 착상들이 수많이 제기되고있다. 그리고 하드웨어의 다량생산과 그의 질적변화는 다양한 봉사를 만들어내였다. 본문중심의 통보문체계로부터 시각적인 도형과 다매체환경에로의 변화는 망봉사를 개선하고 새로운 봉사분야발전의 기초로 되고있다. 미래의 콤퓨터망을 리용한 활용분야는 대단히 넓어져 사회의 모든 분야에 미치는 영향은 그야말로 크며 이것은 전화의 발견과 발전이 인류문명에 준 영향과 비교하여 볼 때 그에 비할바없이 대단히 크다고 말할수 있다.

콤퓨터망의 기초리론은 다른 분야에 비해 볼 때 얼마 되지 않는다고 할수 있다. 그러나 실지 설치과정에서는 상당히 포괄적이며 임의적인 세부규칙을 따라야 한다. 이러한 세부규칙은 다른 회사의 제품과 호환성을 보장할수 있도록 범용성을 가져야 한다. 우리가 콤퓨터망에 대한 학습을 진행하는것은 단지 리론만을 알자는것이 아니라 실지 실천에서 제기되는 문제들에 어떻게 대처하며 콤퓨터망을 어떻게 구성하여야 하는가에 대하여 알려는데 목적이 있다고 말할수 있다. 콤퓨터망을 원만히 활용하자면 리론에 대한 충분한 인식과 함께 높은 실천능력을 가지고있어야 한다. 망관리자의 경우에는 하드웨어와 쏘프트웨어를 사용하는 능력뿐아니라 망을 리용하여 생산성이 높은 정보흐름을 구성하는 방법도 소유하여야 한다. 또한 망을 구성하는 장치들을 리해하고 쏘프트웨어와 하드웨어의 갱신과 그 성능에 대한 파악에 기초하여 제기되는 문제점을 신속히 찾고 해결하여야 한다.

망설계자의 경우에는 사용자의 요구와 새로운 봉사의 부가가치를 알아야 하며 망에서의 봉사와 앞으로의 발전방향에 대하여서도 잘 알아야 한다. 망은 끊임없이 발전하는 분야이다. 그것은 이 분야가 콤퓨터관련기술들과 련관되여있으며 아직 개선되여야 할 부분이 많은 분야이기때문이다.

1.1.1. 일반적개념

콤퓨터망(Network)이란 용어적인 의미로 볼 때 《망상의 조직》이라는 의미를 가지고있는데 전문가적인 견지에서 해석하면 자료통신 그자체를 말하며 말단(terminal)장치들간의 통신경로를 구성하는 자원의 집합을 말한다. 여기서 자료통신이란 사용자들사이에 실질적인 정보교환을 이루는 모든것을 포괄하여 말하는것이다. 콤퓨터망은 처음에는 값비싼 장치(례를 들어 인쇄장치, 구동기, 매 콤퓨터의 기억매체)를 공유하여 사용하기 위한 것이였다.

그러나 지금은 하드웨어와 쏘프트웨어의 눈부신 발전으로 세계를 련결하는 망이 구축되였으며 그를 리용하여 사용자들사이에 파일이나 통보문 등의 교환을 진행하고있다. 현

재는 음성 및 영상신호를 포함한 다매체정보도 망을 리용하여 전송되고있으며 이를 토대로 하여 세계적 범위에서의 영상전화회의도 진행되고있다.

흔히 망이라고 하면 많은 사람들이 LAN(Local Area Network)을 생각하는데 WAN(Wide Area Network), VAN(Value Added Network) 등 분류방법과 구성방식에 따라 여러가지로 분류할수 있다.

현재 우리가 말하고있는 망을 간단하게 정의하면 사용자들사이의 정보교환을 가능하게 하는 하드웨어와 쏘프트웨어의 결합이라고 말할수 있다.

우리는 이미전부터 망환경에서 생활하며 일하고있다. 실례로 전화망은 우리에게 가장 친숙해지고 널리 퍼져있는 음성전달목적의 망이라고 할수 있다. 비록 단방향이기는 하지만 TV나 라지오도 일종의 망이라고 볼수 있으며 하나의 전기제품의 내부도 전기의 흐름을 조종하는 회로로 구성된 하나의 망이라고 할수 있다.

그렇기때문에 망은 하나의 정의로부터 시작된 학문이였고 특정한 사람들만이 할수 있는 특별한 분야였지만 오늘의 정보산업시대에서는 빠르고 정확한 정보의 공유라는 측면에서 볼 때 망이 사람들의 일상생활에까지 널리 리용되는것은 필연적인것이라고 볼수 있다.

오늘의 현실로부터 알수 있는바와 같이 콤퓨터망이 없었다면 정보기술혁명이나 정보산업시대의 도래는 생각도 할수 없었을것이다. 이러한 리유로 하여 콤퓨터망은 경제와 문화 등 사회 여러 분야의 발전을 추동하는 기둥으로 등장하고있으며 그것이 차지하는 몫은 앞으로 더욱 커질것이다.

1.1.2. 발전과정

콤퓨터망의 발전력사에 대하여 이야기하자면 먼저 전기통신의 발전력사로부터 시작하여야 할것이다. 1830년경 사무엘 모르스에 의해 전신기가 발명되여 처음으로 전기에 의한 통신이 실현되였다. 1876년 벨에 의해 전화기가 발명되였다. 최초의 전화체계는 사용자와 사용자가 직접 련결되여있는 점대점(point to point)방식이였다.

이 방식은 1880년대에 와서 교환수가 수동으로 교환해주는 교환체계로 발전하였다. 전자식교환기는 1890년대에 와서야 개발되였으며 지능화된 교환기는 1970년대에 도입되기 시작하였다. 그 다음 수자식전송방식이 개발되고 이것이 ISDN(Integrated Services Digital Network)의 기초로 되였다.

우의 그림에서도 알수 있는것처럼 오늘의 콤퓨터망의 발전은 곧 전기통신의 발전이라고 볼수 있다. 여기서 전화나 라지오, TV가 인간생활에 널리 도입되게 된것은 그것들이 사람들사이의 정보교환에 리용되였기때문이라고 볼수 있다. 마찬가지로 콤퓨터망의 리용에서도 기본핵심은 정보의 공유이다. 정보의 공유를 떠나서는 오늘날 세계를 뒤덮은 인터네트(Internet)에 대해서 생각하지도 못했을것이며 정보산업의 시대라는 말은 생겨나지도 못하였을것이다.

이러한 현실적조건들과 정확한 정보를 보다 빨리, 그리고 구체적으로 얻으려는 사람

들의 요구로 하여 콤퓨터망은 앞으로도 계속 발전의 길을 걸을것이다.

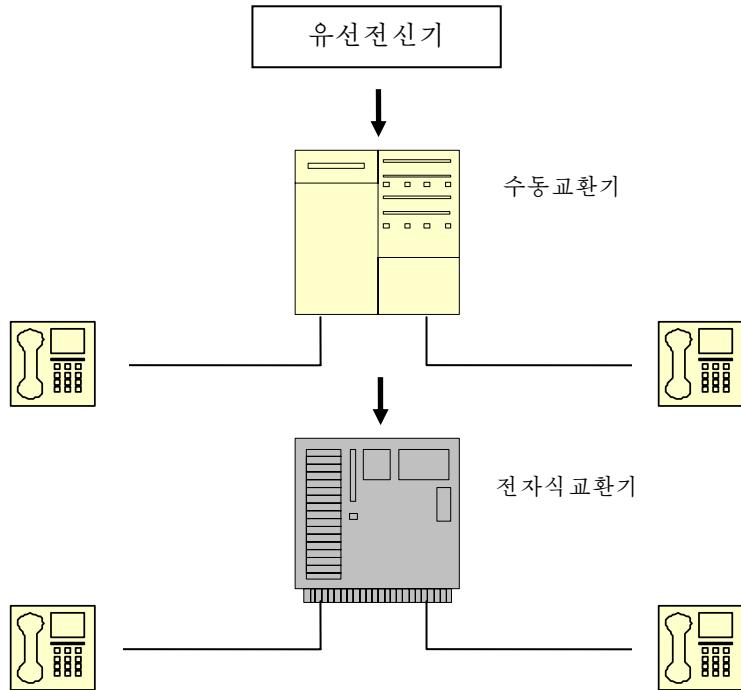



그림 1-1. 전화기의 발전력사

1.1.3. 구성요소

콤퓨터망을 단순하게 분류하여 보면 송신장치, 전송매체, 수신장치로 나누어 볼수 있다. 콤퓨터망의 기능은 크게 이 세가지중의 하나로 표현된다. 이 세가지요소를 통털어 통신장치련결(Communication Link)이라고 하며 이 요소들은 콤퓨터망에서 서로 공유된다고 말할수 있다.

통신장치련결의 공유는 보통 콤퓨터망에서 중추망(Back Bone)이라는 기술에 의하여 이루어진다. 사람에 비유하여 보면 중추라는것은 우리 몸에서 척추를 이루는 뼈들을 말한다.

우리 몸에는 수많은 신경들이 무수히, 그리고 복잡하게 퍼져있는데 이러한 신경들은 모두 뇌로 향하고있다. 하지만 모든 신경들이 다 뇌수에 련결되여있는것은 아니다. 말단신경들은 중추신경으로 뻗어있으며 이 중추신경들이 모아져 뇌로 이어진다.

사람의 몸은 그야말로 정교한 콤퓨터망을 구성한다고 말할수 있다. 다시말하여 모든 기관의 신경망들이 뇌로 향하는 중추망을 공유하고있는 형태라고 보면 리해하기가 쉬울것이다. 의학에서 중추신경망과 기능신경망의 분리를 보여주는 실례가 바로 무릎의 반사신경을 알아보는 실험이다. 작은 망치로 무릎의 가운데 부분을 치면 본인의 의사에는

관계없이 무릎이 올라가게 된다. 여기에 대한 의학적인 해석은 무릎에서 받은 충격에 대한 신경반응이 중추망을 거쳐 뇌로 전달되는것이 아니라 중추신경에서 거기에 대응하는 신호가 무릎으로 되돌아가는 현상이라고 설명된다. 이와 류사한 현상은 콤퓨터망에서 볼수 있는데 파케트(packet)가 콤퓨터망의 봉사기에 도달하지 못하고 지역경로기에서 다시 송신측에 보내여지는 ICMP(Internet Control Message Protocol)통보문에 비유할수 있다.

여하튼 통신장치련결의 목적은 하나의 공유중추망을 구축하여 각 지역사이의 련결을 실현하는것이다. 만약 모든 통신장치련결이 객체들사이의 직접적인 련결로 된다면 객체가 추가될 때마다 그러한 련결은 기하급수적으로 늘어나게 될것이다. 이것을 허용한다면 세계를 련결하는 콤퓨터망의 구축은 실현불가능한 일로 된다. 이러한 문제점을 해결할수 있게 해준것이 바로 중추망과 지역망이다. 이것은 우에서 사람의 신경을 놓고 설명한것과 같은 원리에 기초한것이므로 어렵지 않게 리해가 될것이다.

통신장치련결의 공유와 함께 중요하게 제기되는 또 다른 문제는 공유된 자원을 모든 의뢰기(Client)들과 말단들이 합리적으로 공평하게 리용하는것이다. 다시말하여 중앙에 위치한 봉사기(Server)는 매개 의뢰기들의 요구를 모두 공평하게 처리하여야 한다는것이다. 만약 봉사기가 특정한 말단이나 의뢰기에 대해 독점적인 봉사만을 허용한다면 그것을 제외한 많은 의뢰기들은 그 봉사가 끝날 때까지 기다려야 할것이며 그것은 사용자들에게 마치 의뢰기가 멎은것처럼 보일수 있다. 이 문제는 얼핏 생각하여 보면 해결방도가 잘 떠오르지 않는다. 한마디로 어떤 사람에게 여러 사람이 질문을 하고있는데 그 사람은 질문에 대한 대답을 주는데서 매 사람의 물음에 어떻게 대답을 주는가 하는 문제라고 볼수 있다. 한명씩 대답을 주면 명백하고 좋겠지만 한 사람에게 대답을 주는동안 다른 사람들은 대답이 끝날 때까지 기다려야 한다. 또한 누구의 질문에 먼저 대답을 주겠는가도 매우 어려운 문제이다. 결론부터 본다면 한사람씩 상대해서는 모든 사람의 요구를 다 들어줄수 없다는 결론이 나온다. 콤퓨터망의 자료공유에서도 역시 같은 문제가 제기된다.

이러한 문제를 해결하기 위해 콤퓨터망에서는 전송되는 자료를 일정한 크기의 단위로 잘게 나누어 취급하는 방법을 리용하고있다. 즉 매개 의뢰기들의 요구를 실시간적으로 갈라서 처리해주는 방법을 말한다. 이 방법은 여러가지 우점을 가지고있지만 여기에도 일련의 결함이 있는데 제일 문제로 제기되는것은 잘게 나누어진 파케트의 류실이다. 즉 전송과정에 이러저러한 원인으로 하여 전송되는 자료들이 없어지게 되는 경우도 있는데 이러한 문제점으로 하여 전송되는 다른 모든 자료의 믿음성들이 떨어지게 된다.

송신장치(Transmitter)는 말그대로 수신측으로 보낼 자료를 만들어내여 전송에 필요한 부가자료 즉 식별부호, 오유검출 등의 정보들을 붙이거나 암호화하는 장치를 말한다.

전송매체(Transmission Medium)는 송신측과 수신측을 련결하여주는 물리적인 매체를 말한다. 이 매체에는 재질과 속도, 매체의 특성에 따라 여러가지로 나눌수 있는데 보통 동축케블, 동선, 빛섬유 등이 있고 무선통신에서는 대기공간을 념두에 두기도 한다.

수신장치(Receiver)는 송신장치로부터 받은 전송정보와 자료를 구별하여 재조립하는

장치로서 송신측의 오유검출사항과 꼭같은 코드를 가지고 전송자료의 오유여부를 판별한다. 그리고 만약 오유가 검출될 경우 재전송을 요구하여 자료의 정확성을 보장한다.

1.1.4. 오유조정

콤퓨터망상에서 오유가 발생하는 원인은 주로 물리적인 전송매체의 불량, 통신량의 증가로 인한 자료전송의 교착상태, 비루스, 잘못된 전송상태의 설정 등이다. 이러한 오유에 대처하기 위하여 수신측에서는 수신파케트에 붙어오는 전송정보를 해석하거나 정해진 전송시간을 초과하였을 경우 송신측에 재전송통보문을 보낸다.

전송정보의 해석단계에서는 두가지 방법중 한가지를 리용하여 해석을 진행한다. 그중 한가지는 긍정확인(Positive Acknowledge)인데 이 경우 수신측은 파케트를 정확하게 수신한 경우에만 송신측에 확인통보문을 보낸다. 만약 전송오유가 많이 발생할것같은 콤퓨터망에서 긍정확인방법을 사용한다면 많은 효과를 볼수 있다. 다른 한가지는 부정확인(Negative Acknowledge)인데 이는 수신측에서 받은 파케트의 오유정보를 계산하여 오유라고 판단되였을 경우 송신측에 재전송을 요구하는 방법이다. 일반 콤퓨터망들에서는 우의 긍정확인방법과 달리 망이 믿음성이 있을 경우 이 방법을 쓰면 효과적이다. 이러한 오유조종에 대해서는 많은 알고리듬이 존재하고 콤퓨터망을 설치할 때 설정이 가능하기때문에 콤퓨터망관리자나 설계자는 사용하려는 콤퓨터망의 상태를 파악하고 적절한 오유조종을 설정하여야 한다.

1.1.5. 다중접근(Multiple Access)

콤퓨터망은 망상에서 공유되는 자원들(례를 들면 봉사기의 하드디스크에 저장된 파일들, 콤퓨터망상의 인쇄기, 자료기지의 자료들, 봉사기를 통해 련결할수 있는 다른 토막의 콤퓨터망 등)에 대한 다중사용자의 접근 등을 기본으로 리용하고있기때문에 다중접근과 관계되는 콤퓨터망설계에 대한 리해가 있어야 한다.

국부망(Ethernet)의 기초로 되는 ALOHA체계로부터 시작하여 다중사용자의 련결을 통한 콤퓨터망자원의 공유라는 개념은 콤퓨터망의 가장 중요한 특징이다. 우리가 흔히 볼수 있는것처럼 보통 콤퓨터에서 기억기에 적재된 일정한 주소의 자료에 대하여 다른 두가지 장치(프로그람)에서 동시에 접근하려고 하면 공유오유통보문이 나타난다.

또한 자료기지상에서 하나의 표에 보관되여있는 정보에 대하여 서로 다른 두대이상의 의뢰기가 동시에 갱신하려고 하면 오유가 발생한다. 이러한 오유상태에 대하여 봉사기의 자료조종장치는 공유위반사항에 대해 어떠한 규칙을 적용하여 판단한다. 콤퓨터망에서도 이러한 문제를 조절하는 어떤 법칙이 있는데 이것은 보통 규약이라고 부르는 일련의 약속과 규칙에 의해 조종된다.

상식

ALOHA 란 무엇인가?

ALOHA 는 하와이말로 ≪어서 오십시오.≫라는 인사말이다. ALOHA 콤퓨터망은 무선콤퓨터망인데 하와이대학에서 하와이를 둘러싸고있는 각 섬들과 통신을 진행하기 위해 만든 콤퓨터망으로서 1970 년대 초에 만들었다고 한다. 송신과 수신주파수가 다르고 (전송은 407MHz, 수신은 413MHz) 9600bps 의 속도로 파케트를 전송한다. 오늘의 시대에서 9600bps 는 대단히 느린 속도지만 그 당시에는 굉장한 연구성과였다. 때문에 많은 연구소에서 관심을 가지고 ALOHA 를 연구하였는데 Palo Alto 연구소에서 ALOHA 를 기초로 하여 Ethernet 를 개발하였다.

ALOHA 콤퓨터망과 Ethernet 콤퓨터망의 공통점은 다수의 콤퓨터가 망에 접근할수 있는 다중접근콤퓨터망이라는것이다. 동시에 여러 말단이 콤퓨터망에 접근하는 경우 일정한 지연시간(실지로는 란수에 기초한 시간)이 지난 후에 접근을 다시 시도한다. 모든 마디들은 콤퓨터망상에서 다른 사용자의 접근을 감시한다. 이것을 콤퓨터망용어로 청취(Listen)라고 한다.

ALOHA 는 제일 처음 개발된 콤퓨터망으로서 하나의 통신통로에만 접근할수 있다. 하나의 통로가 통신중이면 다른 통신통로는 임의의 시간동안 대기하여야 한다. 사실 ALOHA 의 이러한 원리는 Ethernet 와 같지만 Ethernet 는 ALOHA 가 통신통로를 공유하는데 비해 일정한 시간을 매개 마디에 분할해서 접근을 허용하는 시분할체계라는것이다.

1.1.6. 규모에 따르는 분류

LAN(local area network)이라는 말은 간단히 국부망이라고도 하는데 흔히 근거리통신망을 의미하는것으로써 사무실의 한개 층이나 학교 또는 제한된 지역에 설치하여 독립적인 각종 장치들을 서로 접속시켜 통신할수 있도록 구성된 콤퓨터체계의 총칭이다.

WAN(wide area network)이라는것은 광대역망이라고도 하는데 LAN은 하나의 사무실이나 건물에 국한된다고 하면 WAN은 하나의 도시를 념두에 둔다고 볼수 있는데 보통 사용자가 1000명이상 되는 경우를 말한다. 그렇기때문에 주콤퓨터에 걸리는 부담이 크므로 초고속처리가 가능한 콤퓨터를 사용하여야 한다.

WAN의 체계구성은 보통 몇개의 LAN들을 모아 고속전송이 가능한 회선을 주콤퓨터에 접속시키는 형태로 구성한다. 이러한 LAN의 중심에는 처리기능마디를 설치하여 주콤퓨터의 부담을 줄인다. 보통 WAN은 매개 나라의 전기통신기관이 운영제공한다.(그림 1-2)

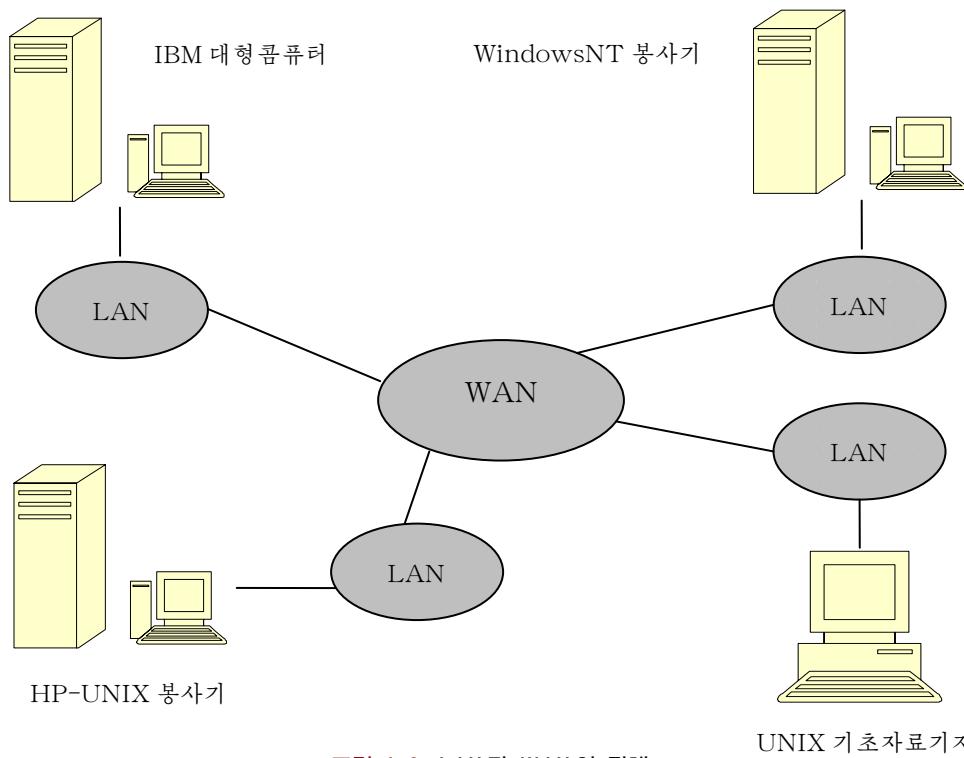



그림 1-2. LAN과 WAN의 관계

VAN(value added network)은 부가가치통신망이다. 1970년대에 어느한 나라에서는 전기통신봉사기관들로부터 고속수자식전용회선을 빌리여 저속통로로 분할하고 빌려주는 통신회선의 재판매가 시작되였다. 이때 단순한 재판매를 인정하지 않고 어떠한 부가가치를 붙일것을 요청하였다. 이로부터 VAN이라는 용어가 생겨 났다. VAN은 이렇게 통신회선을 빌려 콤퓨터통신망을 구축하고 이 통신망으로 통신봉사기관이 제공하지 않은 봉사를 제공하는 일을 말아한다.

규모에 따라서는 우와 같이 크게 3가지로 갈라 볼수 있다. 우선 LAN만 보아도 Ethernet, Tokenring, Tokenbus, FDuI(Fiber Distributed Data Interface), DQDB (Distributed Queue Dual Bus) 등 종류가 많다.

1.1.7. 파케트

파케트는 콤퓨터망상에서 전송되는 자료의 기본단위로서 사용자의 자료에 머리부와 꼬리부라는 오유탐지비트가 덧붙어져 수신측에서 받은 송신측자료를 검증할수 있도록 만든것을 말한다. 이러한 검증기능이 있었기때문에 콤퓨터망을 통한 자료의 전송에 널리 리용되여 발전할수 있었다.

파케트자료는 크게 정적자료와 동적자료로 나눈다. 정적자료는 비트파일로 변환이 된다. 즉 0과 1로 구성된 하나의 파일로서 보조보관장치에 정적인 상태로 존재할수 있다는

것을 말하는데 이것은 비동기식통신방법의 객체로 된다. 동적자료는 비트렬로 변환된다. 즉 이것은 기억기상에 존재하거나 정적자료의 구성요소로 되며 이것은 동기식통신방법의 객체로 된다.

1.1.8. 봉사에 따르는 분류

봉사에는 먼저 동기식봉사(Synchronous Services)가 있다. 이때 동기란 시간적인 관계가 일치되여 있는것을 말한다. 다시말하여 동기식봉사란 자료의 기본이 되는 단위인 비트를 나타내는 각 신호의 발생시점이 고정된 시간기준에 관계되는 자료전송을 말한다. 동기식봉사로 자료를 전송하면 전송되는 비트렬은 모두 동일한 지연시간을 가지고 전송된다. 다시말하면 첫번째 자료비트가 목적지에 도착한 시간과 마지막 자료비트가 목적지에 도착한 시간이 동일하다는 뜻이다. 따라서 비트마다 시간을 신호로 사용하여 송신하고 수신측에서 시간신호에 의해 비트마다 자료를 수신하는 방식을 동기식전송이라고 한다.

동기식통신(Synchronous Communication)봉사는 일정한 지연(Delay)과 오유비률로 비트렬을 전송한다. 즉 어떤 비트들은 정확하게 전송되지 않을수 있지만 비트렬을 구성하는 모든 비트들이 일정한 지연후에 전송된다.《동기식》이라는 말은 모든 자료가 동시성을 가진다는것을 의미한다. (그림 1-3)

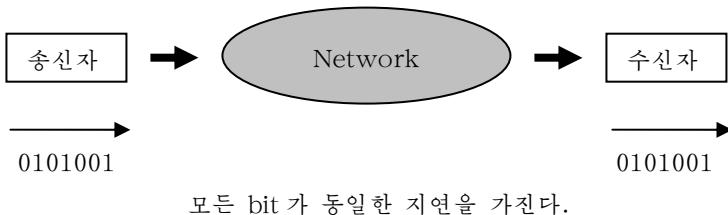



그림 1-3. 동기식봉사

실례를 들면 우리가 전화를 걸 때 잡음도 섞이고 예상치 못한 혼선이 생길 때가 있는데 이러한 잡음과 혼선에 대해 일정한 지연을 가진다는것은 동시성을 의미하는것이다. 즉 송신과정에 오유가 생겼다고 해서 전화를 거는 사람의 목소리가 수신측에 아주 늦게 들리거나 전혀 다른 목소리로 들리는것은 아니다.

다음으로 비동기식봉사(Asynchronous Services)가 있는데 이것은 문자 또는 그 이상의 적당한 단위 즉 블로크의 선두를 탐지한 순간을 기준으로 하여 시작비트와 정지비트에 의해 송수신동작을 합치는 방식으로서 비동기식전송이라고 한다.(그림 1-4)

비동기식통신(Asynchnorous Communication) 자료(정확히는 비트들)들은 파케트로 나누어지는데 전송될 파케트들은 서로 다른 지연시간을 가지게 된다. 이 파케트들은 수신측에서 완충기에 저장되며 다시 일련의 순서를 가지게 된다. 여기서 완충기는 특별히 보조저장장치보다 빠른 기억기에 일정한 저장공간을 확보하여 입출력을 빠르게 할 목적으로 만

드는것이다. 중간에 완충기를 두는것이 얼마나 유용한가 하는것은 누구나 조금만 생각하면 쉽게 리해가 될것이다.

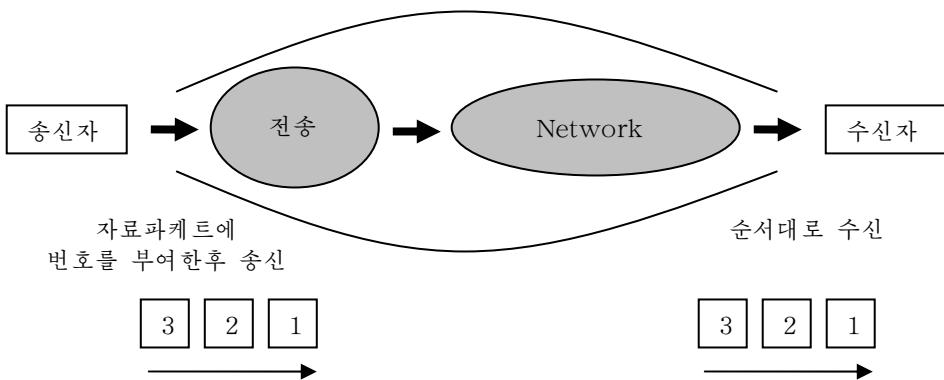



그림 1-4. 비동기식봉사

비동기식통신봉사에 영향을 미치는 요소들은 오유비률, 지연시간, 송신측의 믿음성과 보안성 등이다. 비동기식통신봉사로 동기식봉사를 흉내 낼수는 있는데 이것의 실례가 파케트음성봉사이다

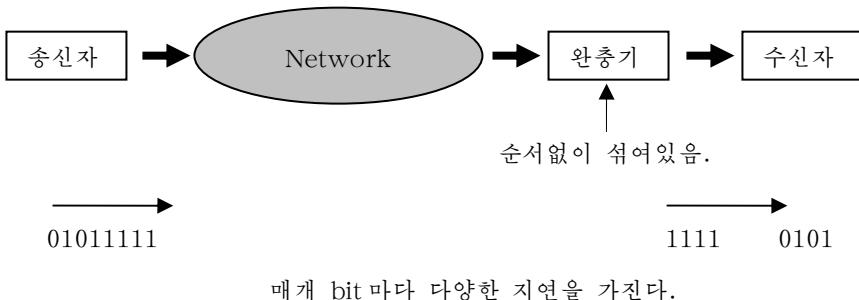



그림 1-5. 접속지향형 봉사

비동기식봉사는 접속지향형봉사와 비접속지향형봉사로 분류한다.

접속지향형봉사(Connection Oriented Services)는 일단 접속을 설정해놓고 순서대로 파케트를 전송하는것이다.(그림 1-5) TCP(Transmission Control Protocol)규약 같은 봉사가 접속지향형봉사에 속하는데 이것은 안정된 가상경로를 설정하고(송신측과 수신측의 1대1 경로) 파케트를 정확한 순서대로 전송하며 파케트의 손실을 허용하지 않는 안정된 전송규약이다. 우편에서 등기를 생각하면 리해를 쉽게 할수 있다.

비접속지향형봉사(Connectionless Oriented Services)는 파케트를 순서에 관계없이 개별적으로 전달한다. 따라서 송신측이 보낸 자료에 대한 담보가 없으며 수신측은 수신된 자료를 순서에 따라 재배렬 및 재조립하는 과정을 거쳐야 한다. 따라서 때로는 오유를 그대로 가질수도 있고 자료의 일부가 손실될수도 있다.(그림 1-6)

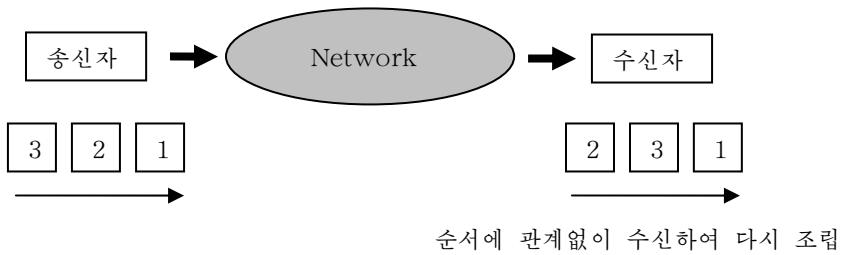



그림 1-6. 비접속지향형봉사

비접속지향형봉사는 접속지향형봉사보다는 좋게 말하면 융통성이 있다고 할수 있고 나쁘게 말하면 얼떨떨한 봉사라고도 할수 있다. 제각기 우결함을 가지고있지만 비접속지향형봉사는 융통성에 중점을 둔 봉사이다. 이 봉사는 가상경로를 설정하지도 않으며 모든 파케트에 순서도 없을뿐만 아니라 송신시 여러 경로로 다중형변화를 진행한다. 이때 손실되는 자료에 대해서는 송신측이 책임을 지지 않는다.

이렇게 정확하지 못한 측면이 있는 반면에 속도면에서는 접속지향형봉사보다 우월하므로 정확한 전송을 요구하는 자료봉사에는 쓰이지 않고 주로 짧고 신속히 진행되여야 하는 봉사에 쓰인다. UDP(User Datagram Protocol)가 이러한 봉사에 속하는데 주로 전자대화나 통보문봉사 등에서 리용한다.

또 다른 봉사로서 지급봉사가 있는데 이 봉사는 파케트에 우선권을 부여하는 방식으로 동작한다. 지급자료봉사는 급히 보내야 할 파케트를 전송대기중인 파케트들의 가장 앞부분에 놓아서 전송을 진행한다.

지금까지 여러가지 통신봉사에 대하여 보았다. 모든 통신봉사를 우의 분류에 포함시킬수 있지만 최근의 봉사들은 서로 절충하거나 제거 및 통합하여 보다 최적화된 통신봉사를 제공한다.

자기가 사용하여야 할 통신봉사의 내용에 대한 파악은 콤퓨터망을 리해하고 콤퓨터망상에서 제기되는 이러저러한 문제들을 해결하는데서 실마리를 주는 중요한 요소중의 하나이다.

1.1.9. 교환방식에 따르는 분류

자료교환방법은 모두 서로 다른 특성을 가지고 실현되는데 공통된 목적은 사용자가 제한된 하드웨어자원을 공유하도록 접속성을 가지게 하는데 있다. 이전의 단일과제체계와는 달리 최근의 Windows계렬의 조작체계나 OS/2, UNIX 호환조작체계들은 선점형다중과제처리방식으로 작업을 한다. 그리고 다중스레드(multithread)라는것도 프로쎄스(process)를 보다 작게 나누는 방법으로 커져가는 화상지향프로그람들의 기억기 및 CPU에 대한 합리적인 공유를 보장한다. 례를 들면 우리가 word처리기를 다중과제, 다중스레드 환경의 조작체계

에서 실행시켜보면 실행파일이 기억기에 적재될 경우 하드디스크가 회전하는 소리를 들을 수 있다. 하지만 이 실행파일이 기억기에 적재된다고 해서 모든 word프로세스의 기능이 기억기상에서 구현되는것이 아니다. 일정한 기능을 위해 아이콘을 찰칵하면 다시 하드디스크가 돌아가는 소리를 들을수 있다. 이것은 실행파일이 그 기능에 해당한 동적련결서고파일을 호출한다는것을 말하며 모든 word처리기의 모듈이 기억기에 적재되는것이 아니라 매개 기능의 스레드가 차례로 기억기에 적재된다는것을 의미한다.

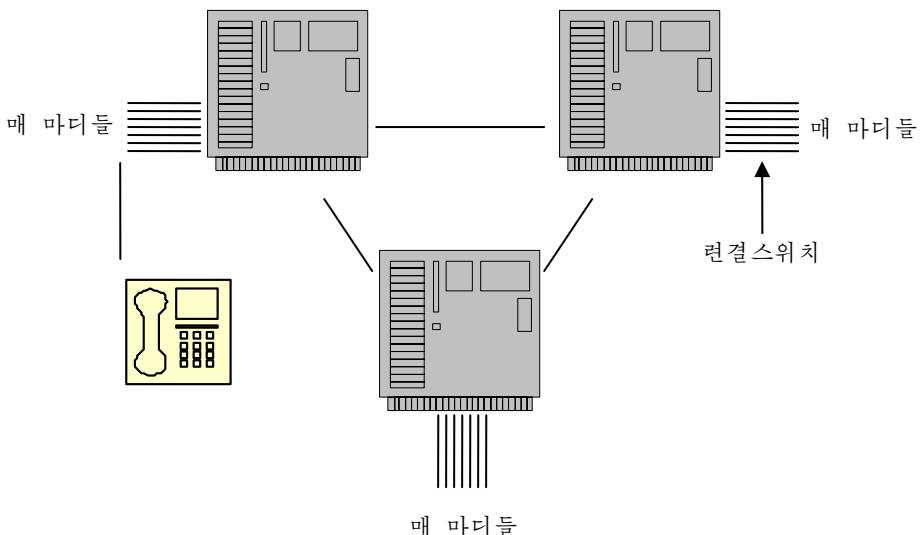



그림 1-7. 회선교환봉사

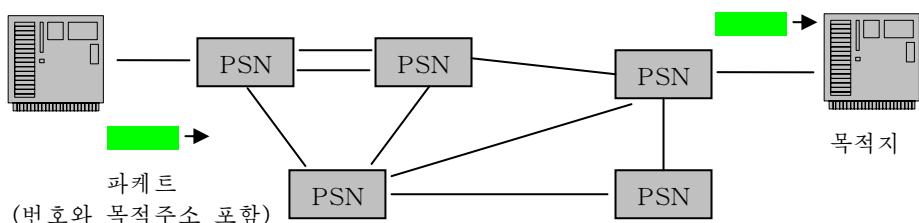



그림 1-8. 파케트교환봉사

교환방식에 따르는 분류에는 먼저 회선교환봉사(Circuit Switched Services)가 있는데 봉사를 요구할 때마다 두개이상의 자료말단장치를 접속하여 그 접속이 해제될 때까지 그것들사이의 자료회선을 전용으로 사용하도록 하는 방식이다. 즉 콤퓨터망에서 회선을 공유하는 기술이다. 회선은 사용자가 리용하는 동안에는 독점되며 그것이 해제되여야 다른 사용자의 리용이 가능하다.(그림 1-7)

회선교환콤퓨터망은 주로 전화망에서 사용하는 방법인데 전화망은 매개 지역에 따라 교환기지국을 두게 되여있다. 즉 우리는 어느 누군가에게 전화를 걸때 그 수신자와 직접 련결을 가지는것이 아니라 매 지역에 설치되여있는 교환기지국과의 련결에 의하여 통화를 진행하는것이다.

그래서 교환기지국의 교환기는 매 전화사이의 련결을 이루기 위해 많은 련결을 가지게 되는데 이런 련결을 회선교환련결이라고 한다. 그리고 이러한 회선교환련결을 통해 많은 전화들이 통신을 진행할수 있는것이다. 여기서 회선교환망은 공유된다. 회선교환방식의 우점은 일단 련결이 이루어지면 지연이나 오유조정이 크게 필요없다는것이다.

파케트교환방식이 전송의 지연이나 오유가 있을 경우 수신측에서 재전송을 요구하는 반면에 회선교환의 경우 이러한 요구가 없다. 그렇기때문에 수신된 자료의 완전성을 요구할 필요가 없고 전달된 자료의 내용에 대한 담보가 부족하다. 주로 전화기의 음성봉사나 부분적인 손실이 전체 자료에 큰 영향을 미치지 않는 영상봉사 등에 쓰인다.

이와 달리 파케트교환(Packet Switching)방식의 봉사는 상대방에 대해 지정한 파케트를 사용하여 자료전송의 경로를 결정하고 전송하는 처리방식으로서 목적한 파케트의 전송기간만큼 통로가 리용되지만 그 전송이 끝난후에는 다른 파케트의 전송에 의해 통로의 리용이 가능하게 된다. 이 방식에서의 자료전송은 처음 자료가 여러개의 파케트로 분해된다.

매 파케트에는 수신측의 주소와 일련의 번호가 붙는다. 파케트를 수신한 수신측에서는 파케트의 번호를 리용하여 정보비트렬이나 파일을 다시 구성하여 처음과 같은 자료로 복구한다.(그림 1-8)

파케트교환(Packet Switching)콤퓨터망은 회선교환콤퓨터망과 같은 개념이지만 전송방법에 있어서 축적후전송(Stored and Forward Transmission)방법을 리용한다. 축적후전송이란 말그대로 전송되는 파케트를 받은 중간전송장치(Packet Switching Node)가 이 파케트들을 축적한 후에 다음 경로로 보내는것이다.

회선교환장치에서 중간교환기는 다음 경로로 보내는 회송장치에 해당된다. 축적후전송의 실례를 든다면 우리가 편지를 쓰면 그 편지는 우편통에 모아진 후 우편국으로 전송되고 모든 우편통의 편지들은 우편국으로 집결된 다음 다시 목적지의 우편국으로 발송되여 거기서 필요한 곳으로 가게 된다.

파케트교환콤퓨터망은 전송되는 파케트들의 축적될 경로의 설정방식에 따라 크게 가상회선파케트교환방식과 데이터그램(datagram)파케트교환방식으로 나누어진다. 이 두가지 전송방식은 콤퓨터망상의 전송경로를 설정하는 중요한 방법이므로 잘 알아두어야 한다.

먼저 가상회선파케트교환봉사(Virtual Circuit Packet Switched Services)에 대하여 보기로 하자. 이 봉사는 통신을 시작하기전에 상대방과 론리적전송경로의 접속을 설정하고 그 접속를 리용하여 통신을 진행한다.(그림 1-9)

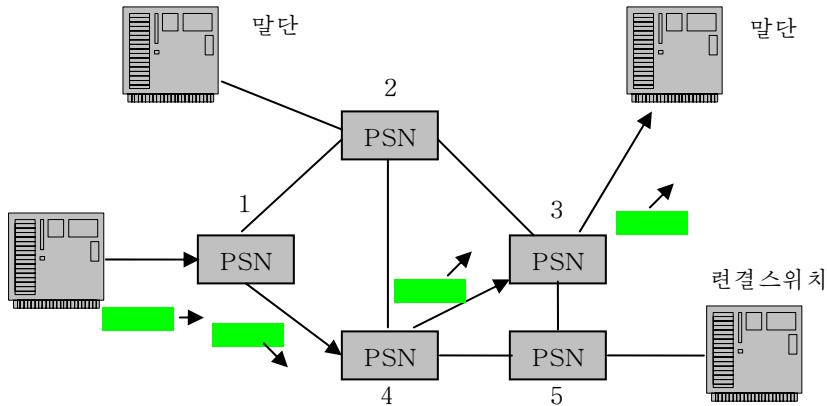



그림 1-9. 가상회선파케트교환봉사

그리고 통신이 끝난 다음에는 접속을 단절시키는 전화형식의 통신형태이다. 통신을 시작할 때 자료전송경로를 확보하면 간단한 규약으로 큰 규모의 자료를 전송할수 있는 우점이 있으나 전송통로를 하나의 프로쎄스가 전용으로 사용하기때문에 상대적으로 비용이 많아지는 약점이 있다. 가상회선파케트교환(Virtual Circuit Packet Switching)은 《가상적》(virtual)이라는 단어가 의미하는것처럼 송신측에서 수신측까지의 전송경로를 가상으로 설정한다.

TCP규약의 경우가 가상회선을 만드는 대표적인 규약으로 유명한데 일단 가상회선이 만들어지면 모든 파케트들은 이 회선을 따라가야 한다. 물론 가상회선안에 다른 송신측에서 보낸 파케트도 뒤섞일수도 있지만 송신측과 수신측의 1:1전용구간을 만드는것이라고 생각하면 된다. 가상회선은 봉사기자료기지의 질문처리 등과 같은 오랜시간의 접속상태에서 짧은 시간동안에 전송을 처리해야 하는 콤퓨터망에서 쓸모가 있다. 은행의 각 지점과 본점간의 자료기지의 갱신작업 등은 호상 오래동안 접속을 유지하면서 봉사기의 자료기지에 질문을 하고 빠른 시간동안에 결과가 나와야 하므로 가상회선파케트교환방식이 아주 적합하다.

이번에는 데이터그램파케트교환봉사(Datagram Packet Switched Services)에 대해 보기로 하자. 이 봉사는 파케트교환에서 미리 정해진 상대방에게 자료를 전송하는 봉사이며 전송과정에 망이 다른 데이터그램을 참고할 필요가 없는 방식을 말한다.(그림 1-10)

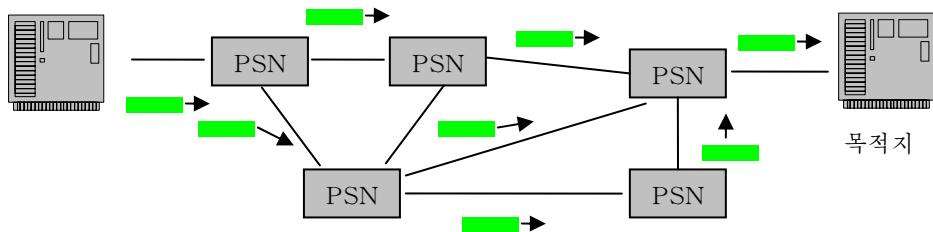



그림 1-10. 데이터그램파케트교환봉사

우점은 어떠한 설정도 필요하지 않다는것이다. 따라서 작은 자료전송에 리상적이다. 또한 매개의 파케트에 대하여 특별한 경로선택이 이루어지므로 마디나 련결의 이상현상에 빠르게 대응하는 경로선택이 가능하다. 그러나 콤퓨터망에서 데이터그램파케트를 전송할 때 일부 파케트가 손실될수도 있다.

참고로 파케트들은 파케트정보안에 TTL이라는 시간값을 가지고있는데 이것은 Time To Live의 략자로서 콤퓨터망상에서 수신측에 도달할 때까지 존재할수 있는 시간을 말한다.

만약 전송중에 통신량의 과잉으로 하여 TTL이 경과되면 수신측에서는 이 신호를 무시하여 버린다. 즉 무의미한 신호가 되여버리는것이다. 이것을 설명하는 리유는 데이터그램파케트교환방식이 가상회선파케트교환방식과는 달리 경로가 일정하지 않으므로 주의해야 하기때문이다. 물론 물리적인 경로가 단일하다면 이러한 파케트교환방식도 가상회선방식처럼 동작하겠지만 전용선이 아닌 경우 즉 공중으로 파케트들이 전송될 경우에는 심각한 문제를 일으킬수 있다.

물론 오유조종을 통하여 재전송설정을 할수도 있겠지만 데이터그램파케트교환방식은 가상회선방식에 비해 볼 때 믿음성이 낮다. UDP(User Datagram Protocol)가 그러한데 UDP의 경우 앞에서도 언급했지만 저장용자료의 전송보다는 주로 통보문의 처리에 사용되고있다. 전자대화프로그람 등과 같이 별로 중요치 않은 자료 즉 간단한 통보문의 교환 등에는 데이터그램파케트교환이 효과적이다.

앞에서 믿음직하지 못하다고 언급하였지만 쓰이는 목적에 따라 우점도 가지고있다. 우점은 콤퓨터망상에서 부하가 적고 속도가 빠르다는것이다. 여러 경로로 분산시키는것은 병렬처리의 우점이기때문에 빠른 전송을 목적으로 하는 그리 중요치 않은 자료의 전송에는 데이터그램전송방법을 많이 쓰고있다.

1.1.10. 망접속형태에 따르는 분류

망접속형태에 따르는 분류는 콤퓨터망의 물리적인 접속형태에 따라 분류한것으로써 여기에는 모선형, 고리형, 별형, 나무형 등이 있다.

모선형(Bus Topology)은 망의 접속구성을 표시하는 망접속형태의 하나인 모선이라고 부르는 통신케블에 수많은 국(말단)들을 분산접속시키는 형태이다. 대표적인 모선형국부망에는 Ethernet나 10BASE5가 있다. 고리형망도 모선케블을 사용하지만 케블을 고리모양으로 사용하기때문에 한계가 없다. 이와는 달리 모선형국부망에는 한계가 있다. 케블을 통하여 전송된 전기신호가 끝에서 반사되여 잡음으로 되는것을 방지하기 위하여 케블의 끝에 저항을 붙여 신호의 에네르기를 흡수한다. 하나의 모선에 접속되여있는 여러개의 국들가운데서 해당한 국에 자료를 전송하기 위하여 매 국들에 주소를 배당하는데 이 주소가 바로 MAC주소이다.

고리형(Ring Topology)은 고리모양의 전송로(Transmission line)에 여러개의 말

단이 접속되여있는 형태이다. 다시말하여 모선이라고 부르는 통신케블을 고리모양으로 하여 수많은 국(말단)들을 분산접속한다. 고리형국부망에서는 집선기를 고리모양으로 접속하기때문에 집선기의 끝은 없고 집선기우에서의 자료전송방향은 한방향이다. 따라서 린접한 국들사이의 거리가 짧은 부분에서는 금속케블을 사용하고 거리가 긴 부분에서는 빛섬유케블을 사용하여 유연하게 망을 구성할수 있고 케블의 길이가 긴 국부망을 실현할수 있다. 대표적인 고리형국부망에는 IEEE802.5가 표준화한 통표고리국부망과 ANSI가 표준화한 FDDI(빛섬유회선자료대면부)국부망이 있다. 고리형망구성은 국부망만이 아니라 넓은 지역에 설치하는 간선망에서도 사용되고있다.

별형(Star Topology)은 주처리기가 중앙에 위치하고 매개의 처리기와 주처리기가 통신을 진행하는 방식을 말한다. 다시말하여 망의 중심으로 되는 마디를 경유하여 말단들이 호상 접속하는 형태를 말한다. 망구성에는 물리적인 구성과 론리적인 구성이 있으며 두가지가 일치하는 경우와 차이나는 경우가 있다. 례를 들어 10BASE-T는 론리적으로 모선형이지만 물리적인 배선은 집선기를 중심으로 하는 별형망이다. 론리적인 구성과 물리적인 구성이 다같이 별형망인것은 PBX(구내교환기)를 중심마디로 하는 망이다. 이 형태의 특징은 한 처리기의 고장이 콤퓨터망전체에 영향을 미치지는 않지만 중앙조종장치가 고장나면 전체 체계가 마비되는 약점이 있다.

나무형(Tree Topology)은 나무가 하나의 뿌리에서 줄기가 나오고 다시 여러개의 가지로 뻗어나간것처럼 하나의 주콤퓨터로부터 여러개의 말단들이 가지처럼 접속되여있는 형태이다. 이런 계층구조를 가진 콤퓨터망은 고리를 형성하지 않으나 약점은 하나의 말단으로부터 다른 말단으로의 경로가 굉장히 길어질수 있다는것이다.

1.1.11. 자료전송시점에 따르는 분류

콤퓨터망을 전송시점별로 분류해 볼수 있는데 먼저 CDMA/CD(IEEE 802.3) 방식에 대해 보도록 하자. 이 방식은 어느 한 회사에서 Ethernet를 사용한것이 출발점으로 되였는데 그후 IEEE에서 LAN의 표준으로 지정했다. 이 방식은 자료흐름의 조종기능을 매개 통신말단에 분산시켜 전송속도를 높이게 함으로써 전송시간을 단축시켰다. 이것은 송신말단이 자료를 송신하려는 순간의 케블상의 자료흐름을 검출하여 만약 충돌이 일어난 경우에는 오차시간만큼 지연하여 자료를 재전송하는 구조로 되여있다.

이번에는 통표체계방식에 대해 살펴보자. 통표체계는 크게 통표고리와 통표모선으로 갈라 볼수 있다. 통표(Token)라는 말에서 알수 있듯이 마디의 콤퓨터망접근은 통표를 소유한 마디만이 가능하다.

이것은 간단히 수건돌리기에 비유하여 볼수 있는데 통표라는 말에서 알수 있듯이 사용자가 통표를 가지고있을 때에만 콤퓨터망을 사용할수 있다. 통표체계가 만들어진 리유는 동시접근시 발생하는 파케트의 충돌을 방지하기 위해서이다. 통표라는 하나의 사용권한신호를 통해 접근마디를 조종하는것이다.

통표고리(Token Ring, IEEE 802.4)는 IBM에서 사용하고 있고 IBM회사에 의하

여 발전된 LAN기술이다. 이 방식은 고리형의 망접속형태를 가지며 콤퓨터망을 구성하는 모든 마디들로 물리적인 실제고리가 형성되고 이 고리주위로 통표라는 특정한 파케트가 회전한다.

통표모선(Token Bus, IEEE 802.5)은 모선형망접속형태이다. 즉 형태는 모선형이면서 자료의 전송시점은 통표라는 파케트를 가진 시점에서 시작하는 형태이다.

1.1.12. 전송방식에 따르는 분류

콤퓨터망을 전송방식별로 분류해 보면 크게 광대역(Broadband)방식과 기초대역(Baseband)방식으로 나눌수 있다. Broadband(광대역전송)방식은 직결대면부로부터의 신호를 모뎀장치를 써서 변조시켜 송신하는 방법을 의미한다. 변조방법에는 주파수변조와 위상변조방법이 있다.

기초대역방식은 신호변조없이 송신하는 방식으로 대면부회로의 조건에 따라 +와 -를 사용하여 신호를 조합한다. 이것은 단지 하나의 신호만을 전송하게 된다.

1.1.13. 그 밖의 콤퓨터망

먼저 빛섬유콤퓨터망인 FDDI(Fiber Distributed Data Interface)에 대해 보도록 하자. FDDI는 다른 IEEE표준들과 달리 ANSI(American National Standard Institude) ATM(Asynchronous Transfer Mode)의 표준이다.

Ethernet의 10Mbps속도와는 달리 빛섬유를 사용하여 100Mbps이상의 속도를 낸다. 이것은 1987년에 개발되였는데 최고 500개의 마디와 접속할수 있고 매 빛섬유의 최고길이는 200Km이며 매 마디사이의 거리가 2Km까지 가능하다. 세계적으로 많은 나라들에서 이것을 사용하고 있다. FDDI의 련결구성은 그림 1-11과 같다.

FDDI는 SMP(Station Management Protocol)라는 FDDI관리규약에 의해 장치들의 고장과 FDDI의 재구성을 관리한다. FDDI는 마치 통표고리와 같은 형태로 구성되여있지만 만약 콤퓨터망상의 어떤 마디에서 고장이 생기면 FDDI의 고리구조는 모선구조로 바뀐다. 이 방식의 우점은 우연적인 마디의 오유에 대처하여 콤퓨터망을 보호할수 있는것이다.

FDDI는 2중고리형식으로 련결되여있기때문에 MAC계층에서 통표를 보낸다. FDDI에서 사용되는 MAC규약을 시간제한통표기법(Timed Token Mechanism)이라고 한다.

비동기전송방식인 ATM(Asynchronous Transfer Mode)에 대하여 본다면 ATM은 대역폭에 대한 다양성, 많은 싸이트들에 도입할 정도의 공인된 안정성, 그리고 기존의 LAN과 광대역망인 WAN과의 통합의 편리성 등의 우점을 가지고있다. 현재 세계적으로 많은 학교, 기관, 기업소들에서 ATM을 도입하고있는데 약점은 비용이 많이 드는것이다.

이 밖에도 많은 종류의 콤퓨터망들이 있다. 그런데 매개 종류의 망들이 다 우결함을 가지고있기때문에 어느것이 더 우월하다고 평가할수는 없다. 그러므로 콤퓨터망을 구성하거나 관리할 때 어떤 목적으로, 어떤 방식으로 구성하고 관리하겠는가를 잘 연구하여야 한다.

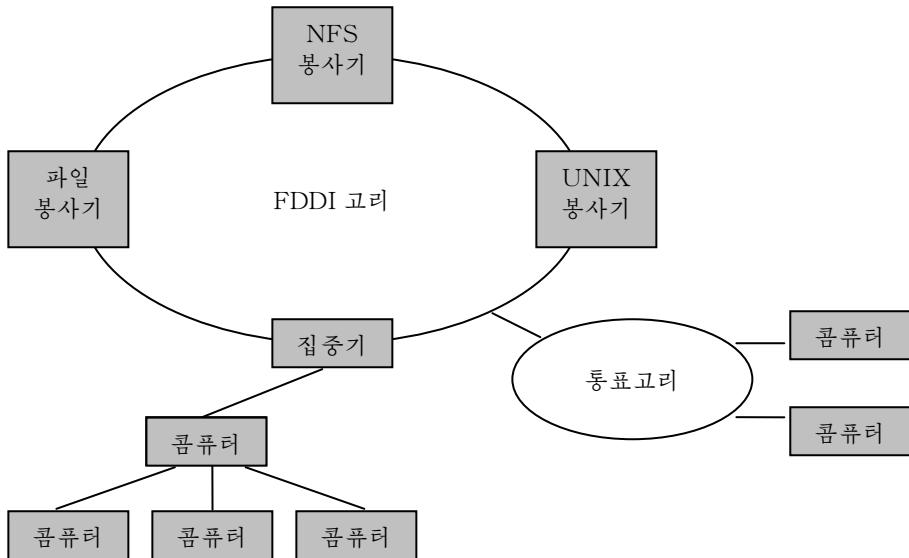



그림 1-11. FDDI 망 구성

제2절. 규　약

1.2.1. 규약이란 무엇인가?

콤퓨터를 약간이라도 다루어 본 사람이라면 TCP/IP규약이라는 말을 들어보았을것이다. 이것은 콤퓨터망개발자이든 아니든 상관없이 자주 듣는 용어인데 여기서는 먼저 규약이란 무엇이며 자주 사용되는 규약에는 어떤것들이 있는가에 대하여 간단히 설명한다.

규약(protocol)이란 콤퓨터망상에서 서로 련결된 체계들사이에 의사소통을 하기 위해 규정한 하나의 약속을 의미한다. 만일 통신객체들사이에 정해진 규정이 없다면 서로가 주고받은 통보문이 무엇을 의미하는지 리해할수 없을것이다. 이러한 규약에는 복잡한 규칙을 가지는것도 있고 아주 간단한 규칙을 가진 규약도 있다.

간단히 실례를 들어 본다면 옛날에 사용하던 봉화도 하나의 규약이 된다고 볼수 있다. 그것은 불이나 연기기둥이 하나인 경우와 둘인 경우 또는 하나도 없는 경우에 따라 서로 무엇을 의미하는지 약속이 되여있으며 이에 따라 정해진 행동을 수행하도록 되여있었기때문이다. 그것을 모르는 사람에게는 그것이 무엇을 말하는지 알수 없으나 미리 약속한 사람들사이에서는 이것을 보고도 서로가 무엇을 말하는지 알수 있었다.

마찬가지로 통신객체들사이에 통보문을 주고받을 때에는 약속된 규약이 먼저 있어야 하며 이 규약에 따라 통보문을 전송하여야 한다.

그리고 통보문을 받는 체계도 규약에 따라 해석을 하고 작업을 해야 한다. 만일 보

내는 쪽과 받는 쪽의 체계들중 어느 한쪽이라도 그것을 지키지 않으면 전송된 통보문은 무시되거나 오동작을 일으키는 요인으로 될수 있다.

상식

콤퓨터망에서 전송되는 자료는 어떤것인가?

콤퓨터망은 전기통신기술에 최근의 콤퓨터통신이 결합된 복잡한 신호전송장치라고 생각할수 있다. 간단히 다른 콤퓨터로 화상이나 본문, 동화상 등을 전송하는것이 콤퓨터망이라고 할수도 있겠지만 사실 콤퓨터망을 통하여 전송되는것은 0 과 1 의 이동일뿐이고 물리적으로 고찰할 때에는 전압의 변화를 통한 신호의 전달이다. 그리고 이러한 물리적인 신호는 항상 매체를 통과하면서 신호에 잡음이 타므로 실지 송수신신호의 변화에 대해서는 예견하기가 어렵다. 우리가 일상생활에서 늘상 느끼는것지만 TV 를 볼 때 허상이 생기는 현상도 이러한 잡음이 타기때문이라는것은 누구나가 잘 알고있는 사실이다.

하지만 지금은 이러한 잡음들도 정보의 수자화를 통해 방지할수 있다. 여기서 수자화라는것은 일정한 구간에서의 련속적인 전압의 변화를 2 진수인 0 과 1 의 조합으로 변환한다는 뜻이다.

콤퓨터망에서의 자료는 항상 송신측과 수신측사이에 존재하기때문에 전송의 정확성에 대한 담보가 무엇보다도 중요하다. 전송의 정확성이 자료의 정확성이므로 이 문제는 콤퓨터망에서 기본문제라고 할수 있다. 콤퓨터에서 다루는 2 진코드는 단순하면서도 (참 아니면 거짓) 일관성이 있고 론리적이기때문에 콤퓨터와 통신이 결합된 콤퓨터망은 자료전송에서 하나의 혁명이라고 볼수 있다.

1.2.2. 규약설계

앞에서 본바와 같이 규약의 의미는 아주 간단하다.

그러나 규약의 의미가 간단하다고 하여 규약의 설계까지 간단한것은 아니다. 물론 단순한 통보문만을 주고받는 문제를 해결하려 한다면 설계라는 개념을 적용하지 않아도 되겠지만 콤퓨터망상에서 복잡한 작업을 수행해야 하는 체계들사이에는 규약의 설계가 간단하지 않다.

콤퓨터망체계를 개발하는 개발자들은 몇가지 규칙에 따라 규약을 설계해야 한다.

첫째로 변하지 않는 골격을 가지도록 만들어야 한다. 즉 새로운 규칙이 추가될 때마다 규약의 전반내용이 변한다거나 다른 규칙이 변하게 되면 안된다. 규약설계시에 간단한

규칙만을 적용해서 규약을 정의하고나면 사용과정에 생각하지 못했던 부분이 추가되면서 변화가 일어날수 있다. 규약이 변하면 이전의 통보문이나 자료가 쓸모없이 되는 경우도 있는데 이것은 매우 심각한 문제이다. 사용과정에 규약의 갱신은 피할수 없는 일이기때문이다. 그러므로 처음 만들 때에는 여러가지 경우를 생각하여 설계를 해야 한다.

둘째로 오유처리를 잘할수 있도록 만들어야 한다. 즉 통보문의 종류와 통보문에 포함된 자료의 내용이 정확히 일치해야 하며 이것을 확인할수 있어야 한다. 그래야 규약을 어긴 통보문인지 아닌지를 알수 있으며 필요한 오유처리를 할수 있기때문이다. 례를 들어 자료를 삭제하라는 통보문인데 새로 작성하여야 할 자료가 포함되여 있다면 이것을 확인한 다음 재전송을 요구하든지 오유처리를 하여야 한다.

셋째로 규약이 너무 커지거나 복잡해지면 안된다. 필요없는 내용을 너무 많이 포함시키거나 애매한 내용이 많이 들어가게 되면 규약으로서의 가치가 떨어지고 이것이 오유를 발생시키는 원인으로 되기때문이다.

이상과 같은 원칙을 지키면서 규약을 설계하여야 한다. 어떤 일이나 마찬가지이지만 규약작성도 설계에 하루를 더 투하하면 나중에 일주일을 절약할수 있다. 그러므로 규약설계를 잘 하여야 한다.

1.2.3. Internet 규약

콤퓨터망통신을 위한 규약에는 종류가 많다. 그 가운데서도 가장 널리 알려져있고 일반화되여 사용되고있는것이 Internet규약 즉 TCP/IP를 리용한 통신규약이다. 그림 1-12는 OSI모형에 기초한 TCP/IP모듈을 표현한것이다.

TCP(Transmission Control Protocol)는 쌍방향통신을 지원하고 byte단위로 자료를 전송하는 련결지향형규약(Connection Oriented Protocol)이다. 대부분의 Internet 프로그람은 TCP를 리용하는데 TCP는 그림 1-12에서와 같이 IP에 의존하기때문에 보통 Internet규약을 TCP/IP라고 부른다.

UDP(User Datagram Protocol)는 비련결규약(Connectionless Protocol)으로서 데이터그램이 목적지에 도달하는가 안하는가에 대하여 알려고하지않는 점이 TCP와 다르다.

그리고 ICMP(Internet Control Message Protocol)는 관문과 주콤퓨터사이의 오유와 조종정보를 조종하기 위한 규약이다. ICMP통보문은 IPC데이터그램을 통해 전달되는데 TCP/IP망쏘프트웨어는 이 통보문을 리용하여 콤퓨터망이 정상인가 아닌가를 확인한다.

하지만 이것은 체계의 내부에서 이루어지는것이기때문에 사용자들은 이것을 몰라도 프로그람을 리용할수 있다. IP(Internet Protocol)는 TCP, UDP, ICMP를 위한 파케트전달봉사를 제공한다. 이때 사용자는 특별히 IP에 대해 신경을 쓰지 않아도 된다. 단지 정확한 IP의 설정만을 하면 된다. 그러면 TCP/IP규약을 기초로 하면서 널리 사용되고있는 프로그람들에 대하여 보기로 하자.

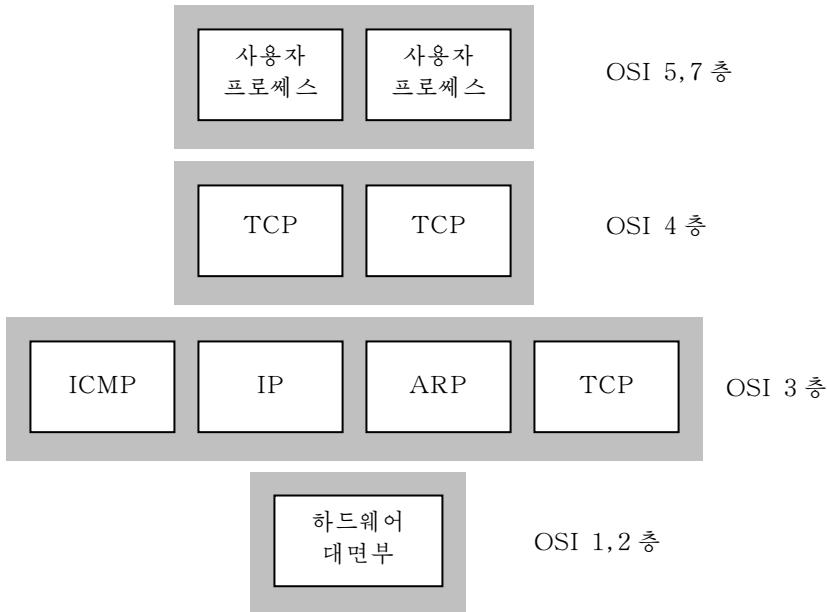



그림 1-12. TCP/IP 모듈

• TELNET: 원격으로 다른 곳의 주콤퓨터체계에 접속하여 주콤퓨터내부의 쉘창을 리용하는것과 같은 효과를 내는 프로그람이다. telnet는 믿음성있는 전송을 목적으로 하는 접속지향형봉사인 TCP대화접속을 리용한다. telnet에서 사용하는 포구는 tcp 23번포구이다. telnet는 본문방식이라는 약점이 있지만 여전히 UNIX계렬의 체계에 접속하여 사용하는 통신프로그람으로 많이 활용되고있다. telnet을 리용할 때에는 적합한 말단방식을 리용하여 접속하여야 한다.

• FTP: 파일전송규약(File Transfer Protocol)의 략자로서 주콤퓨터에서 파일을 얻거나(get) 주콤퓨터에 파일을 전송할 때(put) 많이 사용된다. telnet와 함께 원격으로 다른 곳의 UNIX체계를 사용하는데 가장 많이 리용된다. telnet와 FTP는 서로 보충해주는 특징을 가지고있다. 즉 FTP의 경우에는 주콤퓨터안에서 작업수행이 불가능하지만 파일전송은 가능하며 telnet는 파일전송은 불가능하지만 주콤퓨터안에서 작업수행이 가능하다.

• TFTP: FTP와 같이 통보문전송을 목적으로 하지만 TCP대화접속을 리용하는 FTP와는 달리 UDP대화접속을 리용한다. 따라서 믿음성에서는 문제가 있지만 빠른 파일전송에 적합하다.

• SNMP: 콤퓨터망관리체계에서 많이 사용하는 망관리용규약이다. 여기에 대해서는 뒤에서 자세히 보기로 하자.

• SMTP: Simple Mail Transfer Protocol의 략자로서 단순한 우편전송규약이다. 우편을 송신하는 봉사기로는 SMTP봉사기를 많이 리용한다. TCP대화접속을 리용하는

SMTP규약은 RFC 821에 명시되여있으며 Internet전자우편을 전송하는 엔진으로 사용된다. UNIX에서 많이 사용하는 Sendmail프로그람이 바로 이 규약을 리용하고있다.

• POP: 사무우편물규약(Post Office Protocol)의 략자로서 tcp 109번포구를 사용한다. 최근에 많이 사용하는 POP3은 POP의 발전형으로서 tcp 110번포구를 사용하고있다. SMTP봉사기가 우편을 전송하는 봉사기로 많이 리용되고 있다면 POP봉사기는 우편을 수신하는 봉사기로 많이 리용된다.

제3절. UNIX콤퓨터망

지금까지는 콤퓨터망의 일반적개념과 콤퓨터망관리체계에 대한 개념을 위주로 하여 간단히 보았다. 이 절에서는 이러한 리론들중 UNIX와 관련된 내용을 위주로 보도록 하자. 책의 제목과 다르다고 생각할수 있는데 Linux조작체계가 UNIX조작체계의 한개 갈래라고 할 때 그 리유를 어렵지 않게 짐작할수 있으리라고 본다. 또한 UNIX와 관련된 콤퓨터망부분이지만 기타 다른 조작체계에서도 적용되는 내용들도 많다.

UNIX상에서 콤퓨터망을 다시 정의한다면 정보를 공유하는 체계들의 집합이라고 말할수 있다. UNIX에서는 때로 UNIX체계자체의 처리속도보다 정보를 공유할수 있는 콤퓨터망의 련결여부와 련결속도를 더 중시할 때가 많다.

1.3.1. 콤퓨터망주소

체계호상간에 련결을 설정하려면 서로의 주소를 알아야 한다. 이때 사용되는 주소에는 먼저 Ethernet주소가 있다. Ethernet주소는 물리적 주소 또는 MAC주소라고도 불리우는데 대체로 6Byte로서 콤퓨터망의 말단기의 PROM에 기록되여있다. Ethernet주소는 같은 콤퓨터망안에서 서로 다른 주소를 가지고있어야 한다.

콤퓨터망에 련결된 주콤퓨터들은 각기 자기의 이름을 가지고있는데 이러한 이름은 별명으로 대치할수도 있다. 주콤퓨터이름 또한 동일한 콤퓨터망안에서는 이름이 서로 달라야 한다.

콤퓨터망주소에서 아주 중요한 주소는 역시 IP주소이다. IP주소 역시 동일한 콤퓨터망안에서는 서로 다른 주소를 가지고있어야 한다. 만일 세계콤퓨터망에 련결된 체계라면 전세계적으로 유일한 IP주소를 가지고있어야 한다.

IP주소는 4개의 10진수로 이루어지는데 매 수자는 점으로 분리하여 표시하며 0~255사이의 수자여야 한다. 따라서 체계가 가지는 IP주소는 0.0.0.0~ 255.255.255.255중에 하나가 된다. IP주소는 체계가 속해있는 콤퓨터망을 나타내는 부분과 해당 콤퓨터망속에서 체계를 표시하는 부분으로 나누어진다.

례를 들어 192.168.100.101이라는 IP를 가지고있다고 했을 때 3Byte는 콤퓨터망부분이고 1Byte는 체계부분이다. 즉 192.168.100은 콤퓨터망 번호이고 101은 체계번호이다. 콤퓨터망상에서 해당 체계를 찾을 때에는 처음 3개의 Byte를 리용하여 해당 콤퓨터망을 찾

고 마지막 Byte를 리용하여 해당 체계를 찾게 된다.

그러면 몇Byte까지가 콤퓨터망부분이고 몇Byte가 체계부분인가 하는 물음이 제기된다. 이를 위하여 IP주소는 내부에서 이를 계산할수 있는 정보를 제공한다. 먼저 IP주소는 콤퓨터망부분을 위하여 등급을 나누고있는데 이것을 클라스(class)라고 부른다. 클라스는 아래와 같이 A, B, C클라스가 존재한다. 매개의 의미를 간단히 설명하면 다음과 같다.

• **A클라스:** 첫번째 Byte가 콤퓨터망부분, 나머지 3개의 Byte는 체계부분이다. 따라서 콤퓨터망을 제외한 부분만큼 주콤퓨터에 IP를 부여할수 있다. (255x255x255개만큼)

• **B클라스:** 두번째 Byte까지가 콤퓨터망부분. 나머지 2개의 Byte는 체계부분이다. 따라서 255×255개만큼 콤퓨터망안의 체계들에 IP를 할당할수 있다.

• **C클라스:** 세번째 Byte까지가 콤퓨터망부분이다. 총 255개의 IP를 할당할수 있다.

매개 클라스들은 IP주소의 첫번째 Byte의 제일 높은 자리의 bit를 리용하여 계산을 진행한다. 이것을 간단히 설명하면 A클라스는 첫번째 bit가 0으로 설정된다.

1.3.2. 자료전송

UNIX에서 콤퓨터망을 리용하여 자료를 전송할 때 사용하는 가장 작은 단위는 파케트이다. 파케트는 머리부와 통보문으로 이루어지는데 머리부속에는 파케트를 보내는 체계의 IP주소, 파케트를 수신하는 체계의 IP주소, 그리고 어떤 종류의 통보문이 있는지를 나타내는 정보 등이 포함되여있다. 그리고 머리부를 제외한 통보문부분에는 실지 보내려고 하는 자료의 내용이 포함된다.

파케트를 만들고 전송을 하게 되면 파케트를 발송하는 체계는 파케트의 머리부를 보고 수신체계의 주소를 조사한다. 만일 주소가 내부(local)망에 있는 체계이면 수신체계에 곧바로 파케트을 전송하게 된다. 내부망에 련결된 체계가 아니면 파케트를 경로기로 전송한다.

경로기로 파케트가 전송되면 경로기는 경로배정표에 수신체계가 포함되여있는가를 검사한다음 그것이 있다면 수신체계에 파케트를 넘겨준다. 만일 자신의 경로배정표에서 해당 체계를 찾을수 없으면 다른 경로기에 파케트를 전송한다.

상식

NFS(Network File System)란?

UNIX는 다른 체계에서 사용하고있는 파일체계를 자기의 체계안의 파일처럼 사용할수 있는 NFS 기능을 지원한다. 봉사기로 사용되는 체계는 자신의 파일체계를 멀리에 떨어져있는 체계가 사용할수 있도록 제공한다. 의뢰기로 사용되는 체계는 봉사기에서 제공하는 자원을 자기것처럼 활용할수 있다.

최근에는 Windows에서 UNIX에 있는 파일체계를 NFS로 활용할수 있도록 해주는 Samba 봉사기를 많이 사용하고있다.

1.3.3. 콤퓨터망구축

개발자들이나 사용자들이 동작중인 UNIX체계에 접속할 때에는 많은 경우 콤퓨터망은 사용할수 있도록 설정이 되여있는 상태이다. 만일 콤퓨터망기판에 문제가 있어 체계를 다시 설치하는 경우, 또는 개인이 사용할 목적으로 PC용 UNIX를 설치할 때에는 망카드의 구입으로부터 설치, 설정과정에 이르기까지 모든 과정을 거쳐야 한다. 콤퓨터망설정이 되여있는 경우 아래와 같이 'ifconfig'지령을 리용하여 망대면부가 어떻게 설정되였는지 확인할수 있다.(그림 1-13)

ifconfig의 실행결과를 보면 lo와 eth0 두개의 대면부가 설정되여있는것을 볼수 있다. 여기서 lo는 체계자신을 표시하는것이고 또 다른 대면부인 eth0은 콤퓨터망의 대면부가 별도로 설치되여있음을 보여주는것이다. eth0대면부의 내용을 보면 Ethernet주소가 “00:0C:76:00:DD:4E”이라는것을 알수 있다.

그리고 부분망마스크는 ffffff00 즉 255.255.255.0임을 알수 있다. 그리고 현재 UP상태로 콤퓨터망봉사를 진행하고있다는것을 알수 있다. 만일 콤퓨터망설정을 다시 하려면 ifconfig지령을 리용하여 콤퓨터망대면부가 가진 값들을 변경하면 된다.

그림 1-13. ifconfig의 실행결과

IP주소를 바꾸는 실례를 보도록 하자. 먼저 IP주소를 바꾸려면 현재 실행중인 콤퓨터망봉사를 중지하여야 한다

```
% ifconfig eth0 down
```

콤퓨터망봉사가 중지되였으면 다음과 같은 지령을 리용하여 IP주소를 변경시킨다.

```
% ifconfig   elx0   inet 192.168.8.102
```

그다음 봉사를 다시 시작(up)한다.

```
% ifconfig   eth0   up
```

IP 주소를 변경한 다음에는 원격으로 Ping이나 Telnet 등을 리용하여 IP가 제대로 변경되였는지 검사하여야 한다. IP주소변경외에 netmask를 변경하거나 다른 내용을 변경할 때에도 이와 류사한 방법을 사용한다. 즉 《ifconfig 대면부명, 실마리어, 새로운 값》을 리용하면 된다. 만일 체계의 IP주소를 변경할 때 다음번 체계기동시에도 새로운 IP로 계속 사용하게 하고싶으면 /etc/hosts파일을 변경하여야 한다. /etc/hosts 파일을 vi 편집기 등으로 연다음 localhost로 되여있는 부분을 변경된 IP로 바꾸도록 한다.

콤퓨터망대면부의 설정이 끝났으면 망안에 련결되여있는 경로기와의 련결설정이 제대로 되였는가를 확인하며 외부망과의 련결이 제대로 되는가를 검사한다.

체계안의 경로배정표를 확인하는것도 좋은 방법인데 이때는 netstat 지령을 리용하면 된다. 그림 1-14는 netstat지령을 실행하고 그 결과를 보여준 화면이다.

그림 1-14. netstat의 실행결과

UNIX의 경우 체계자체가 경로기의 역할을 수행할수 있다. 물론 경로기의 역할을 수행하는것이 꼭 좋은것은 아니지만(성능저하 등의 문제로) 이러한 기능을 제공하고있다는 것을 알아두고 필요할 때 활용하면 좋은 점도 있다.

경로기의 설정이 제대로 되였으면 해당한 경로를 통하여 접속을 시도할수 있다.

지금까지의 과정을 통하여 콤퓨터망에서의 전반적인 내용에 대하여 간단히 보았다.

보다 자세한 내용을 학습하려면 전문참고서적을 보아야 한다. 콤퓨터망의 설계와 관리를 깊이 파악하자면 리론에 대한 학습을 보다 깊이있게 하여야 한다.

제2장

프로세스

서론

이 장에서는 Linux조작체계에서 중요한 문제의 하나인 프로세스에 대해 설명한다. 프로세스는 프로그람의 실행단위를 의미하는것으로써 체계에서 실행되고있거나 실행예정 및 실행완료된 모든 프로그람들과 련관되여있다.

이 장에서는 먼저 프로세스란 무엇이며 어떻게 관리되고 동작이 진행되는가 또한 프로세스를 사용하기 위한 체계호출(함수)은 어떻게 하는가에 대하여 설명한다. 이때 사용되는 체계호출은 프로세스의 생성, 실행, 완료 및 관리 등과 관련된 내용들이다. 그리고 이러한 체계호출과 그에 도움이 될만한 암시를 리용하여 프로세스관련프로그람들을 작성하게 된다. 2장의 차례를 간단히 소개하면 다음과 같다.

목표

1. 프로세스 구조
2. 프로세스 체계호출
3. 프로세스 프로그람작성

제 1 절. 프로쎄스 구조

2.1.1. 프로쎄스

여기서는 프로쎄스란 무엇이며 어떻게 생겨나고 없어지는가 하는데 대해 보기로 하자.

프로쎄스란 간단히 정의하면 조작체계상에서 실행되는 개개의 프로그람을 의미한다. 이때 단순히 프로그람이라고 하면 2진파일로 이루어진 기계어들의 모임을 의미할수도 있다. 그러나 프로쎄스는 기계어로 이루어진 수동적인 프로그람이 아니라 체계자원을 할당받아 동작하는 능동적인 프로그람을 의미한다. 또한 프로쎄스에는 CPU의 등록기, 탄창기억기 등 프로그람의 실행에 필요한 자원들을 모두 할당받아 언제든지 자기 차례가 되면 실행이 가능한 형태의 과제들도 포함된다.

Linux조작체계는 다중과제를 지원하는 조작체계이다. 이때 다중과제는 다중처리로 표현할수도 있는데 이것은 여러개의 과제 또는 프로쎄스가 독립적으로 자기만의 령역을 가지고 동시에 실행된다는것을 의미한다. 따라서 특정한 프로쎄스의 비정상적인 실행이나 완료는 다른 프로쎄스에 크게 영향을 미치지 않는다.

프로쎄스들은 호상 쉽게 교신을 할수 없다. 만일 호상교신이 필요하면 프로쎄스들사이의 통신기법을 통하여 서로의 작업내용을 주고받아야 한다. 프로쎄스들사이에 통신이 필요하다는 개념만 보아도 프로쎄스들이 개별적으로 동작하고있다는것을 알수 있다.

이렇게 프로쎄스가 개별적으로 동작하기 위해서는 매개 프로쎄스가 체계자원들을 소유하고 있어야 한다. 그렇지 않으면 필요한 작업을 제때에 원만히 수행할수 없게 된다. 그리유에 대해서는 수많은 프로그람이 동시에 실행되는 경우를 예상하면 쉽게 리해가 될것이다. 하지만 특별한 경우 일정한 프로쎄스가 체계자원을 독점하는 경우도 있다. 프로쎄스가 체계자원을 독점하게 되면 다른 프로쎄스들은 상대적으로 많은 피해를 보게 된다.

물론 아주 중요한 체계의 주프로쎄스가 체계자원을 독점하는것은 그리 나쁜 일이 아니다. 실례로 자료를 수집할 목적으로 Linux를 사용하고있는데 자료수집프로쎄스가 체계자원의 일부만을 사용하여 작업처리를 제대로 못하면 큰 문제로 된다.

그러나 중요치도 않은 프로쎄스가 체계자원을 랑비하고있다면 이것은 전반적인 체계동작에 나쁜 영향을 주게 된다. 그러므로 이러한것들을 확인하여 프로쎄스를 완료시킨다든지 프로쎄스의 우선권을 떨어뜨린다든가하는 작업이 필요하게 된다. Linux는 이를 위하여 프로쎄스단위의 관리가 가능하도록 조작체계준위에서 각종 지령들을 제공한다.

Linux가 다중처리를 제공하고는 있지만 실제로 하나의 CPU는 언제나 하나의 프로쎄스만 처리하게 된다. 따라서 실행중인 프로쎄스를 제외한 나머지 프로쎄스들은 언제나 자기가 CPU를 사용할수 있는 순간이 될 때까지 기다려야 한다. 또한 기다리다가 자기 차례가 되면 CPU와 필요한 체계자원을 활용할수 있어야 한다. CPU의 경우는 여러 프로쎄스들이 자기 차례를 기다리며 대기하고있을 때에도 쉬는 시간이 없이 동작하게 된다. 따라서 다중처리를 제공하는 체계들은 체계가 가지고있는 자원을 보다 효률적으로 리용할수 있게 되여야 한다. 다중처리를 제공하지 않는 경우에는 실행중인 프로쎄스가 완전히 완료되

여야 다른 프로세스가 움직일수 있기때문에 현재의 프로세스밖에는 자원을 활용할수 없다.

그러면 프로세스의 구조에 대하여 보기로 하자.

프로세스를 표현하는 구조체는 task_struct인데 이것은 task벡토르에 보관되여 관리된다. 다시말하여 프로세스가 task_struct형으로 생성되면 이것들은 다시 task벡토르에 들어가게 되며 이속에서 지적자(point)가 움직이면서 현재 실행될 task_struct를 지정하게 된다. 지정된 프로세스는 정해진 시간안에 필요한 작업을 수행한후 다음 프로세스에 실행권한을 넘겨주게 된다.

체계가 관리할수 있는 프로세스의 총 개수는 task벡토르가 수용할수 있는 task_struct의 수와 관련된다. 작업이 수행될 프로세스를 지정하는 지적자는 일정관리기에 의해 움직이게 된다. 일정관리기는 일반적인 프로세스와 실시간프로세스(즉 우선권이 높은 프로세스)의 처리를 다르게 한다.

만일 체계에 중요한 영향을 미치는 새치기가 발생하면 일정관리기는 그것을 처리할 프로세스를 최우선적으로 실행되게 만든다. 해당 작업의 처리가 끝나고나면 다시 원래대로 실행되여야 할 일반프로세스에 우선권을 넘긴다.

프로세스는 내부에 상태정보를 가지고있으며 이것을 리용하여 현재 프로세스의 상태를 반영하게 된다. 일정관리기는 프로세스의 상태정보를 리용하여 실행이 가능한 프로세스를 선별하는 작업도 수행한다. 프로세스가 가지고있는 상태정보를 간단히 설명하면 다음과 같다.

- Waiting: 대기중인(실행을 기다리고있는) 프로세스의 상태를 말한다.
- Running: 실행중인 상태로서 프로세스가 CPU에 할당되였고 자원을 사용하면서 작업중인 상태를 말한다.
- Stopped: 정지상태로서 프로세스가 정지요청을 받았거나 다른 프로세스가 긴급히 실행되여야 할 때 실행중이던 프로세스의 상태는 stopped로 바뀌게 된다.
- Zombie: 프로세스는 이미 실행중지가 되였지만 task벡토르에 여전히 남아있는것을 말한다. 프로세스가 실행되지 않기때문에 죽은 프로세스라고 리해하면 된다.

상식

IEEE 란?

IEEE 는 The Institute of Electrical and Electronics Engineers 의 략자이다. 간단히 전기전자학회라고도 한다. 전기, 전자, 통신, 콤퓨터 등의 분야에 대한 기술자단체이다. 전기전자분야의 학회로서는 세계 최대규모로서 회원수는 30 만명을 넘는다. 전자부분품, 통신용모선 및 대면부, 국부망 등을 대상으로 한 표준화활동도 추진하고있다.

프로세스는 자기를 표현하는 유일한 값인 프로세스 ID를 가지고있다. 쉘상에서 ps지령을 수행할 때 나타나는 PID값이 그것인데 PID를 리용하면 프로세스의 상태를 검색하거나 관리 또는 강제로 완료하는 등의 작업을 진행할수 있다.

제일 먼저 기동한 최초의 프로쎄스를 제외한 나머지 프로쎄스들은 부모프로쎄스(Parent Process)를 가지고있다. 이미 존재하고있는 프로쎄스의 복제과정을 거쳐 새로운 프로쎄스가 생성되기때문에 복제를 실행한 프로쎄스와 복제를 당한 프로쎄스가 있게 된다. 이때 복제되여 새롭게 생성된 프로쎄스가 자식프로쎄스(Child Process)로 되며 본래의 프로쎄스는 부모프로쎄스로 된다. 그리고 같은 부모를 둔 프로쎄스는 형제프로쎄스가 된다.

부모, 자식프로쎄스들사이에는 서로에 대한 지적자를 가지고있다. 이러한 지적자를 리용하여 Linux핵심부는 존재하는 모든 프로쎄스들에 접근할수 있고 동시에 개별적인 관리를 진행할수 있게 된다.

Linux 핵심부중에서 시간과 관련된 핵심부는 프로쎄스의 생성시간과 프로쎄스가 사용한 각종 체계자원에 대한 감시를 수행하게 된다. 이 핵심부를 리용하면 응용프로그람들의 프로쎄스정보에 대한 감시가 가능하게 된다.

2.1.2. 프로쎄스생명주기

체계가 기동하면 모든 프로쎄스들의 부모가 될 프로쎄스만이 실행된다. 이 프로쎄스는 프로쎄스를 관리할 핵심부프로쎄스들을 실행시키고 필요한 초기화를 진행하여 실행목적을 달성하게 한다. 핵심부프로쎄스는 체계초기화를 진행하여 전체 체계가 기동할수 있는 환경을 마련해준다. 또한 핵심부프로쎄스는 체계가 필요로 하는 프로쎄스들의 생성 및 기동을 보장한다. 실례로 사용자가 등록가입을 시도할 경우를 생각하여 이것을 처리해줄 프로쎄스를 기동시키는 작업을 진행한다. 이렇게 새로 생성되는 프로쎄스는 부모가 될 프로쎄스의 통제에 의해 만들어진다.

새로운 프로쎄스는 핵심부방식에서 통제를 받으며 일정관리기가 자기를 실행시켜주기를 기다리게 된다. 설정된 프로쎄스는 앞에서 소개했던 task_struct를 할당받고 벡토르속에 들어가게 된다. 프로쎄스가 설정되면 부모프로쎄스가 가지고있는 체계자원을 자식프로쎄스가 공유하게 된다.

이러한 프로쎄스가 체계자원을 사용할 때에는 다른 프로쎄스들이 접근하지 못하도록 하는데 이때는 체계계수기를 리용한다. 례를 들어 체계자원에 접근한 부모, 자식프로쎄스는 자원을 사용할 때에는 계수기를 높이고 사용을 마치면 계수기를 낮추게 된다. 자원을 공유하는 프로쎄스들이 더 이상 자원을 활용하지 않으면 계수기는 0이 되고 다른 프로쎄스에 대해 접근을 막았던 부분을 해제한다.

핵심부프로쎄스는 생성 및 실행되고있는 프로쎄스들에 대해 CPU사용시간, 생성시간, 자원사용시간 등을 관리하게 된다. 이때 이를 감시하기 위해 박자계수기를 리용하게 되는데 박자계수기가 완료되면 신호를 받아 박자계수기가 완료되였음을 알고 계산작업을 진행하게 된다. 이때 사용하는 신호에는 SIGALRM, SIGVTALRM, SIGPROF 등이 있다.

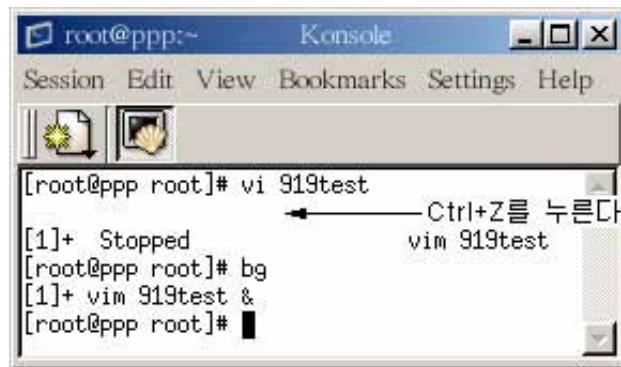



그림 2-1. 프로세스의 중지와 실행실례

프로세스의 실행은 Command Interpreter(지령해석프로그람)에 의해 지령이 해석되고 실행되게 된다. 이러한 지령해석프로그람으로서는 사용자프로쎄스이면서 우리가 잘 알고있는 쉘이다. 쉘은 프로쎄스를 생성하는 fork체계호출을 리용하여 자기의 자식프로쎄스를 복제하게 된다. 복제된 프로쎄스는 쉘이 가지고있는 내용을 자기에게 맞게 변경하고 필요한 작업을 진행한다.

쉘은 복제한 프로쎄스가 실행되는동안 실행을 멈추고 대기하게 된다. 이때 쉘이 동작하기를 바란다면 쉘과 해당 프로쎄스가 다중으로 실행하도록 만들면 된다. 그 과정은 그림 2-1과 같다.

또는 처음에 다음과 같이 실행하면 된다.

```
% runprogram
```

우에서 프로쎄스를 중지시키기 위해 ^Z를 누르면 SIGSTOP신호가 발생되고 프로쎄스는 실행을 멈추게 된다. 그리고 bg지령이나 fg지령을 실행시키면 SIGCONT신호가 발생되며 정지되였던 프로쎄스가 다시 기동한다.

쉘에서 2진파일로 된 프로그람을 실행시키면 해당 파일을 해석할수 있는 해석프로그람프로쎄스가 실행되면서 파일속에 있는 지령어를 실행하게 된다. 만일 쉘프로그람작성을 통하여 만들어진 각본(script)을 실행할 때에는 각본안에서 이것을 해석할수 있는 쉘프로그람을 지정해야 한다. 그러면 쉘각본을 실행하면서 동시에 해당 프로쎄스가 실행되여 각본안의 지령어들을 해석하고 실행하게 된다. 2진파일의 경우에 원천파일을 콤파일하면 실행가능한 ELF형식의 파일이 생성된다. ELF는 Executable and Linkable Format의 략자로서 련결과 실행이 가능한 형식을 의미한다.

ELF형식으로 이루어진 2진파일은 내부에 련결정보를 가지고있기때문에 지령해석프로그람이 지령을 해석하면서 필요한 련결파일의 내용을 함께 실행할수 있다. 그리고 ELF형식의 파일은 필요한 부분만을 기억기에 적재하여 사용하는데 만일 부모프로쎄스가 필요

한 정보를 이미 기억기에 올려놓았으면 그것을 그대로 사용하기도 한다.

프로쎄스는 두가지 방식으로 실행되는데 사용자방식과 체계방식으로 나눌수 있다.

Linux에서는 특정한 프로쎄스가 다른 프로쎄스를 강제로 완료시키거나 정지시키지 못하기때문에 모든 프로쎄스들은 일정한 간격으로 실행을 보장받게 된다. 그러나 모든 프로쎄스가 언제나 자원이나 시간 등을 균등하게 분배받도록 하는것이 좋은것은 아니다. 아주 중요하거나 급히 처리해야 할 프로쎄스인 경우 그것을 다른 프로쎄스들의 처리가 끝날 때까지 기다리게 하는것이 어떤 결과를 가져오겠는가에 대해서는 쉽게 리해하리라고 본다. 그러므로 Linux는 중요하거나 긴급하게 처리해야 하는 프로쎄스와 일반적인 프로쎄스를 구별하기 위해 사용자방식과 체계방식으로 갈라서 실행되도록 한다. 사용자방식으로 실행되는 프로쎄스는 체계방식으로 실행되는 프로쎄스에 비하여 우선순위가 낮다.

프로쎄스는 처음 실행시에는 일반적으로 사용자방식으로 실행되다가 체계새치기나 신호를 통해 체계방식으로 변경된다. 따라서 프로쎄스가 어떤 방식으로 실행되겠는가는 미리 정해지는것이 아니며 체계가 필요에 따라 방식을 변경시켜주기때문에 프로쎄스의 실행이 효률적으로 이루어지게 되는것이다.

프로쎄스 일정관리

체계는 실행우선권에 따라 실행하여야 할 프로쎄스를 선발하여 실행시키는 작업을 일정관리기에 분담하고있다. 일정관리기가 내부의 일정관리규칙에 따라 진행하는 일련의 작업을 일정관리라고 하는데 이것은 조작체계안에서 매우 중요한 작업의 하나로 된다. 그것은 실행을 기다리는 프로쎄스들이 많이 대기하고있는 상태에서 우선권에 따라 프로쎄스를 골라내여 실행하도록 하는것은 안정한 조작체계가 갖추어야 할 필수적인 사항으로 되기때문이다. 따라서 일정관리기는 프로쎄스사용자에게 도움이 되도록 합리적인 규칙에 따라 일정관리를 해야 한다.

Linux는 우선권이 높은 체계프로쎄스와 우선권이 낮은 일반적인 프로쎄스로 나누어서 일정관리를 하게 된다. 만일 우선권이 높은 프로쎄스가 대기렬에 있으면 먼저 꺼내서 실행시킨다. 같은 우선권을 가진 프로쎄스들에 대해서는 FIFO규칙에 따라 실행을 시키게 된다. FIFO는 First In First Out의 략자로서 Q처럼 먼저 기다리고있는 프로쎄스가 먼저 실행되도록 하는 규칙을 의미한다.

프로쎄스가 생성될 때 우선권이 배정되는데 우선권에 따라 프로쎄스의 실행계수가 설정된다. 프로쎄스의 실행시간단위를 턱값이라고 하는데 프로쎄스가 실행에 들어가면 한번의 턱값에 따라 실행계수가 줄어들게 된다. 실행계수가 0이 되면 프로쎄스는 실행을 멈추고 다시 대기렬에 들어가게 된다.

프로쎄스는 대기렬에 있다가 차례에 따라 실행되기도 하지만 신호나 새치기에 의해 실행되기도 한다. 이때는 프로쎄스의 상태가 실행(RUNNING)상태로 바뀌게 되고 일정관리기는 재빨리 해당한 프로쎄스를 찾아서 실행시키게 한다.

일정관리기는 현재 실행중인 프로쎄스보다 먼저 실행을 시켜야 할 프로쎄스를 발견하면 현재 실행중인 프로쎄스를 중단시키고 해당 프로쎄스를 실행시켜야 한다. 이때 실행

중이던 프로세스를 저장해야 할 필요가 있기때문에 프로세스의 정보를 구조(structure)에 보관해서 작업내용이 없어지는 현상 등을 방지해야 한다.

이렇게 작업중인 프로세스의 교체를 상황(context)절환이라고 하는데 다중과제처리를 지원하지 않는 체계의 경우 이와 같은 작업은 체계를 통채로 바꾸는것과 동일한 효과를 가진다. 프로세스는 마지막 시점에서 정지된 상황자료를 체계에 보관하게 된다.

그 다음 다시 프로세스가 실행될 때에는 마지막에 보관했던 상황자료를 다시 적재하여 실행시킴으로써 작업이 중단없이 이어지도록 만들어준다. 만일 프로세스가 자원을 사용하던중이였다면 필요에 따라 자원을 련속해서 사용할수 있도록 만들어준다.

알아둡시다

다중프로세스체계(multi processing system)와 다중처리기체계(multi processor system)는 서로 다른 의미이다. UNIX는 다중처리기와 다중프로세스를 다 같이 지원하는데 이것은 여러개의 CPU가 각기 개별적으로 다중프로세스를 지원한다는것을 말한다.

프로세스가 기동하면서 필요한 작업 및 계산을 진행하는 중앙처리장치를 처리기라고 한다. 처리기는 흔히 알고있는 CPU를 말하는데 이 말은 프로세스실행관점에서 사용하는 용어이다. 처리기를 얼마나 많이 지원하는가에 따라 단일처리기체계와 다중처리기체계로 나눌수 있다. Linux는 여러개의 처리기를 지원하는 다중처리기체계이다.

다중처리기체계는 매개의 처리기들이 다중프로세스를 지원해야 하기때문에 제각기 일정관리기를 가지고있다. 만일 다중처리기인데 하나의 일정관리기만 가지고있다면 다중처리기로써의 성능을 높이기 힘들다.

Linux체계에서는 특별한 작업이 없을 때 idle프로세스가 동작을 하게 되는데 다중처리기인 경우에는 처리기마다 idle프로세스를 가지고있게 된다. 이것은 매개 처리기들의 개별적인 동작 및 작업수행을 지원하기 위해서이다. 하지만 이러한 개별작업들이 때로는 체계의 성능을 저하시킬수도 있다.

실례로 A처리기에서 작업을 진행하던 프로세스가 동작을 멈추었다가 B처리기에서 동작을 시작하게 되는 경우 잘못하면 처음 실행되는 프로세스가 진행한 과정을 다시 거치게 할수도 있다. 간단히 말하여 상황을 새롭게 배치하고 프로그람실행계수를 재설정하는 과정 등을 다시 거쳐야 한다면 이것은 자원의 랑비로 된다. 때문에 일정관리기는 프로세스의 ID를 리용하여 될수 있는 한 빨리 프로세스가 마지막으로 실행되였던 처리기에서 계속 작업을 진행하도록 도와준다.

프로세스와 파일체계

프로세스는 파일(file)체계에서 사용하고있는 방법으로 파일에 접근(access)하게 된다. 즉 파일에 대한 접근권한을 가지고있으면 해당 파일에 접근할수 있지만 그렇지 않으면 파일에 접근할수가 없다. 따라서 이러한것들을 검사하기 위한 값들이 필요한데 이를 위해 프로세스는 내부정보에 GID(Group ID)와 UID(User ID), 그리고 프로세스를 실행

시킨 사용자의 ID를 가지고있다.

프로세스는 여러 사용자와 그룹에 속할수 있는데 만일 소속된 그룹들중 하나가 필요한 파일에 대한 권한이 있으면 프로세스는 해당 파일에 접근할수 있다. 하지만 속해있는 사용자나 그룹이 해당 파일에 대한 권한이 전혀 없다면 필요한 작업을 진행할수 없게 된다. 따라서 프로그람개발자는 이러한 사항을 고려하여 프로그람을 작성해야 한다.

만일 특권사용자계좌만이 사용가능한 체계를 개발했는데 일반사용자계좌로 체계를 사용한다면 어느때인가는 체계가 비정상적인 동작을 할수 있다. 그런데 이러한 사연을 전혀 모르는 상태에서 원인을 찾으려면 전혀 다른 결론에 도달하거나 원인을 발견하지 못하는 경우가 있다. 이런 경우에 대처하여 UNIX개발자들은 체계를 사용할 대상에 대한 관리도 할수 있게 해야 한다.

체계에서 기동중인 데몬들중에는 프로세스내부의 GID/UID를 현재 사용중인 사용자의 GID/UID로 변경시키는 프로세스도 있다. 이것은 데몬이 처음에는 특권사용자의 권한을 가지고 실행되였다고 하더라도 그것을 사용하는 사용자의 GID/UID로 바꾸어줌으로써 그 사용자가 가지고있는 권한에 대한 제한을 그대로 적용시키기 위해서이다. 이러한 과정을 통하여 체계의 보안이나 안전성을 강화할수 있다.

프로세스는 내부에 파일과 관련된 구조체인 fs_struct와 files_struct를 가지고있다. 이 구조체에는 프로세스에 대한 inode와 프로세스가 사용하는 파일들에 대한 정보 등이 들어있다. 이것을 통하여 프로세스가 필요로 하는 파일의 열기/닫기/조사 등이 가능하게 된다. 프로세스가 내부적으로 사용하는 파일을 열게 되면 files_struct구조체는 해당 파일에 대한 지적자(point)정보를 가지게 된다.

프로세스는 가상기억기를 리용하게 되는데 이것은 부족한 기억기를 더욱 효과적으로 사용하기 위해서이기도 하지만 매개 프로세스에 특정한 기억기공간을 할당하기 위해서이다. 이것을 리용하면 기억기를 보호할수도 있고 프로세스들의 독립성을 더욱 높여주는 효과도 거둘수 있다. 따라서 프로세스가 실행될 때에는 필요한 가상기억기를 할당받는 과정을 거치게 된다. 가상기억기속에는 프로세스가 원하는 자료가 적재되게 되며 또한 필요한 서고정보가 적재되게 된다. 만일 동적서고를 사용하게 되면 실제 내용이 올라가지는 않고 서고를 사용할수 있는 련결정보가 적재되게 된다.

이렇게 프로세스들에 개별적으로 할당된 가상기억기안의 정보는 해당 프로세스가 실제로 실행될 때 체계기억기안으로 옮겨진다. 이를 요청페지교체 (Demand Paging)라고 하는데 이것은 필요한 순간에 실지 기억기에 적재되는 기법을 의미한다.

이러한 작업이 가능하려면 프로세스마다 할당된 가상기억기의 공간에 대한 정확한 정보가 체계에 의해 관리되여야 한다. 이를 위하여 프로세스가 가지고있는 가상기억기에 대한 구조체인 vm_area_struct 목록을 OS가 관리하게 된다. 따라서 프로세스의 생성과 실행시 vm_area_struct목록은 갱신되게 되며 기억기의 할당을 위해 끊임없이 참고하게 된다.

제2절. 프로세스 체계호출

이 절에서는 프로세스와 관련된 프로그람을 작성하기 위해 사용할수 있는 각종 체계호출에 대해 보도록 한다.

2.2.1. Fork

프로세스의 생성에 사용되는 함수로는 fork()가 있다. fork()함수는 Linux에서 제공하는 가장 기본적인 프로세스생성함수로서 프로세스 ID를 반환해준다. fork를 통해 핵심부는 프로세스의 복제본을 만드는 과정을 진행하게 되며 새로운 프로세스가 만들어지게 된다.

새롭게 만들어진 프로세스는 fork()를 실행한 프로세스로부터 복제되는데 이때 fork()를 호출한 프로세스는 부모프로세스가 되고 복제된 프로세스는 자식프로세스가 된다. 자식프로세스가 만들어진 다음에는 부모프로세스와 자식프로세스가 동시에 실행되면서 다중과제처리가 이루어진다.

fork()함수가 실행되면 부모프로세스와 자식프로세스는 fork()함수의 다음 문장부터 실행된다. fork()함수의 간단한 사용실례를 보면 다음과 같다.

```
int processID;
processID = fork();
```

우의 실례에서 보여준바와 같이 fork()함수는 정수형의 프로세스 ID(PID)를 반환하게 되는데 이렇게 반환되는 PID를 리용하여 프로세스들을 구분할수 있다. 례를 들어 우의 코드가 실행되였을 때 프로세스 ID에 새로 생성된 PID를 가지고있는 프로세스는 부모프로세스가 된다. 그리고 PID의 값이 초기값인 0을 가지고있는 프로세스는 자식프로세스가 된다.

그 리유는 부모프로세스는 fork()함수를 호출한 결과값을 가지게 되지만 자식프로세스는 fork()함수의 호출결과 어떠한 값을 반환했는지 알수가 없기때문이다. 그리고 만일 fork()함수가 0보다 작은 값을 반환하게 되면 이것은 새로운 프로세스의 생성에 실패하였다는것을 의미한다.

그러면 fork()를 리용한 실례를 보도록 하자. 다음의 실례는 fork()를 리용하여 자식프로세스를 생성한후 부모프로세스와 자식프로세스가 각기 무한순환을 돌면서 통보문을 화면에 출력하는 실례프로그람이다.

실례프로그람: ex_fork.c	
1	#include <stdio.h>
2	
3	/* 무한순환에서 사용할 Forever 선언 */

4	#define FOREVER;;
5	
6	/* 부모프로쎄스가 실행할 함수 */
7	void forParent(void)
8	{
9	/* 실행계수기초기화 */
10	int parentCount = 0;
11	
12	/* 무한순환의 실행 */
13	for (FOREVER)
14	{
15	/* 실행계수기출력 */
16	printf ("parent process - count: %d\n" ,parentCount);
17	parentCount++;
18	
19	/* 3s간 sleep 수행 */
20	sleep (3);
21	}
22	}
23	
24	/* 자식프로쎄스가 실행할 함수 */
25	void forChild(void)
26	{
27	/* 실행계수기초기화 */
28	int childCount = 0;
29	
30	/* 무한순환실행 */
31	for (forever)
32	{
33	/* 실행계수기출력 */
34	printf ("child process-count: %d\n" ,childCount);
35	childCount++;
36	

37	/* 5s간 sleep 수행 */
38	sleep (5);
39	}
40	}
41	
42	/* 프로그람의 주함수 */
43	int main()
44	{
45	/* 자식프로쎄스의 pid용 변수 */
46	int childPID =0;
47	
48	/* fork()함수실행, 자식프로쎄스의 생성 */
49	childPID = fork();
50	
51	/* 자식프로쎄스이면 forchild()함수 실행 */
52	if (childPID == 0)
53	{
54	printf (“<<자식프로쎄스 생성>>\n”);
55	}
56	/* 부모프로쎄스이면 forParent()함수 실행 */
57	else if (childPID > 0)
58	{
59	printf (“<< 부모-자식프로쎄스번호: %d >>\n, childPID);
60	}
61	/* 프로쎄스의 생성에 실패한 경우 */
62	else
63	{
64	printf ("프로쎄스의 생성에 실패했습니다.\n");
65	}
66	return 0;
67	}

그림 2-2. ex_fork의 실행결과

우와 같이 원천파일을 작성하고 해당한 도구를 리용하여 콤파일을 진행하고 실행시키면 그림 2-2와 같은 결과를 얻을수 있다.(프로그람에 대한 구체적인 설명은 프로그람에서 설명부분을 리용하여 진행하였으므로 여기에서는 더 언급하지 않는다.)

프로그람의 실행결과를 보면 자식프로쎄스는 3254번이라는 PID를 가지고 생성되였으며 부모프로쎄스와 자식프로쎄스가 각기 무한순환을 돌면서 통보문을 화면에 출력하는것을 볼수 있다. 만일 다중과제처리를 지원하지 않는다면 우와 같이 두개의 프로쎄스가 동시에 무한순환하는것은 불가능할것이다. ex_fork프로그람을 실행시키면서 다른 쉘창에서 ps지령을 리용하여 fork를 검색해보면 아래와 같이 두개의 프로쎄스가 실행하고있다는것을 확인할수 있다. 이때 3296번을 가진 프로쎄스는 3289번으로부터 파생된 프로쎄스라는것을 알수 있다.(그림 2-3)

그림 2-3. 프로쎄스생성결과 확인

2.2.2. Exit

앞에서 fork()함수를 리용하여 프로쎄스를 생성하는 과정을 보았는데 이번에는 프로

쎄스를 완료시키는 과정을 보도록 하자. 프로쎄스의 생성과정이 중요것처럼 프로쎄스의 완료도 매우 중요하다. 프로쎄스를 완료할 때 사용하는 함수는 exit()함수이다. exit()는 완료의 상태로 사용할 정수값을 인수로 받은 다음 해당한 프로쎄스가 완료되게 하며 완료되면 그것을 체계에 알려주게 된다.

exit()함수의 사용방법을 간단히 서술하면 아래와 같다.

```
int 완료상태;
exit (완료상태);
```

프로그람의 실행과정에는 exit()를 사용하지 않아도 프로그람의 마지막부분에 도달하면 프로쎄스는 자동적으로 끝나게 된다. 하지만 다중과제로 작성된 체계인 경우에 프로쎄스의 작업을 끝낼 때에는 신호처리와 exit()를 함께 리용하여 작업을 완료시키는것이 좋다. 신호가 발생하면 신호에 맞게 완료여부를 판단한 다음에 필요한 작업을 수행하고 프로쎄스를 완료하면 되기때문이다. (신호를 리용하여 프로쎄스를 완료하는 방법은 뒤에서 보기로 하자.)

그러면 이번에는 exit()를 리용한 실례를 보도록 하자. 다음의 실례는 부모프로쎄스와 자식프로쎄스를 각기 exit()를 처리하는 프로그람으로서 일정한 시간간격을 두고 차례로 프로쎄스를 완료시킨다.

실례프로그람: ex_exit.c	
1	#include <stdio.h>
2	
3	/* 프로그람의 주함수 */
4	int main()
5	{
6	
7	/* for문에서 사용할 변수선언 */
8	int step =0;
9	
10	/* 자식프로쎄스의 pid용 변수 */
11	int childPID =0;
12	
13	/* fork()함수 실행, 자식프로쎄스의 생성 */
14	childPID = fork();
15	

```
16        /* 자식프로세스이면 2번 실행후 완료 */
17        if (childPID == 0)
18        {
19                printf ("<<자식프로세스>>\n");
20                for (step=0; step<2; step++)
21                {
22                        printf("자식실행회수:%d\n",step);
23                        sleep(1);
24                }
25                exit(1);
26        }
27        /* 자식프로세스이면 3번 실행후 완료 */
28        else if(childPID > 0)
29        {
30                printf ("<<부모프로세스>>\n");
31                for(step = 0; step < 3 ; step++)
32                {
33                        printf ("부모실행회수:%d\n",step);
34                        sleep (1) ;
35                }
36                exit(1);
37        }
38        /* 프로쎄스생성에 실패한 경우 */
39        else
40        {
41                printf ("프로쎄스의 생성에 실패했습니다.\n");
42        }
43        return 0;
44 }
```

우의 프로그람을 보면 sleep을 리용하여 생성된 자식프로쎄스가 2s후에 완료되며 부모프로쎄스는 3s후에 완료된다는것을 알수 있다. 이 코드를 콤파일하고 실행하면 다음과 같은 결과(그림 2-4)를 얻을수 있다.

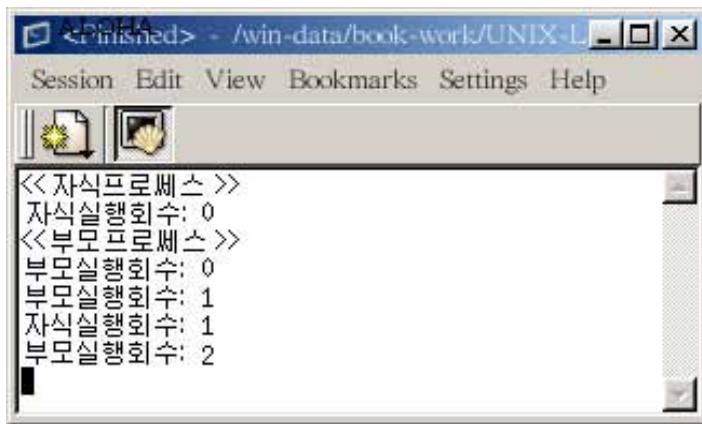



그림 2-4. 프로쎄스의 탈퇴(완료)결과

실행결과에서도 알수 있는것처럼 ex_exit를 실행하면 체계에 처음에는 프로쎄스가 두개 기동하게 된다. 그러다가 하나씩 완료된다. 이때 ps지령을 리용하여 체계상에 있는 프로쎄스를 검사하면 다음과 같은 결과(그림 2-5)를 얻을수 있다.

우의 ps결과를 보면 609번인 부모프로쎄스가 있고 610번이라는 자식프로쎄스가 있다. 그러다가 610번은 완료되고 부모프로쎄스인 609번만이 남은것을 확인할수 있다.

그림 2-5. 프로쎄스완료결과 확인

2.2.3. Exec

프로쎄스가 실행된 다음 실행중인 프로쎄스를 다른 특정한 프로쎄스로 대치시키려고 할 때 사용하는 함수묶음으로서 exec()가 있다. 이때 exec는 이러한 작업들을 수행하는 함수그룹을 대표하는 이름이다. 다시말하여 execl(), execv(), execlp(), execvp() 등 exec로 시작되는 이름을 가진 함수들이 다양하게 존재하는데 모두 같은 작업을 처리하는 함수들이다.

인수로 받아들이는 변수들이 다르기때문에 다양한 함수들이 존재하는것인데 매개 함수들이 사용하는 인수들은 다음과 같다.

- execl: 실행파일의 경로와 인수들을 설정, 마지막에 null을 설정하여 사용

char *path, *arg0, *arg1,····, *argn;
int result = execl(path, arg0, argl,····, argn, (char*) 0);
• execv: 실행파일의 경로와 인수들의 배렬을 사용
char *path, *argv[ ] ;
int result = execv(path, argv) ;
• execlp: 파일이름과 인수들을 설정, 마지막에 null을 설정하여 사용
char *file, *arg0, *arg1,····, *argn;
int result = execlp(file, arg0, arg1,····, argn, (char*) 0) ;
• execvp: 파일이름과 인수들의 배렬을 사용
char *file, *argv[ ];
int result = execvp(file, argv);

exec가 실행되면 프로세스는 exec속에 들어있는 프로세스로 완전히 대치된다. 프로세스가 새롭게 대치된것이기때문에 부모, 자식사이의 관계가 없으며 exec를 호출한 프로세스의 정보를 그대로 계승받는다.

따라서 exec를 실행한 후에도 프로세스의 개수에는 변화가 없으며 새로 실행된 프로세스는 자신의 코드만 실행하기때문에 exec다음에 있는 프로그람부분은 실행되지 않는다. 만일 exec가 제대로 수행되지 않으면 exec다음의 프로그람부분이 실행되기때문에 거기에 오유처리부분을 넣어서 사용하면 된다. 그러면 간단한 실례프로그람을 작성해보도록 하자. 아래는 프로세스가 실행된 다음에 "ps -ef"로 대치되는 실례를 보여주고있다. 이때 사용하는 함수는 execl()함수이다.

실례프로그람: ex_exec1.c	
1	#include <stdio.h>
2	
3	/* execl의 주함수 */
4	int main()
5	{
6	printf ("ps -ef를 실행합니다.\n");
7	/* execl을 통해 ps -ef를 실행 */
8	execl("/bin/ps", "ps", "-ef", (char *) 0);
9	
10	/* 오유발생시에 표시될 통보문 */
11	printf ("오유가 발생했습니다.\n");
12	exit(1);
13	}

프로그람을 콤파일하고 실행하면 다음과 같은 결과가 나온다. 결과를 보면 ps -ef를 실행한것과 결과가 같다는것을 알수 있다. 그리고 오유와 관련된 통보문이 화면에 나타나지 않은것을 통하여 프로쎄스가 대치된 이후에 원래 상태로의 복귀 등이 없이 그대로 완료되였다는것을 알수 있다.

```
<Finished> - /win-data/book-work/UNIX-LINUX/work/2/ex_exec1 - Konsole
Session  Edit  View  Bookmarks  Settings  Help

ps -ef 를 실행합니다.
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  2 16:35 ?        00:00:04 init
root         2     1  0 16:35 ?        00:00:00 [keventd]
root         3     1  0 16:35 ?        00:00:00 [kapmd]
root         4     1  0 16:35 ?        00:00:00 [ksoftirqd_CPU0]
root         9     1  0 16:35 ?        00:00:00 [bdflush]
root         5     1  0 16:35 ?        00:00:00 [kswapd]
root         6     1  0 16:35 ?        00:00:00 [kscand/DMA]
root         7     1  0 16:35 ?        00:00:00 [kscand/Normal]
root         8     1  0 16:35 ?        00:00:00 [kscand/HighMem]
root        10     1  0 16:35 ?        00:00:00 [kupdated]
root        11     1  0 16:35 ?        00:00:00 [mdrecoveryd]
root        15     1  0 16:35 ?        00:00:00 [kjournald]
```

그림 2-6. ex_exec1.c 의 실행결과

실례에서 사용했던 execl()함수를 execv()함수로 대신하려면 아래의 코드를

```
execl( "/bin/ps" , "ps" , "-ef" , (char *) 0);
```

다음과 같이 수정해주면 된다.

```
char *argv[3] ;
argv[0] = "ps" ;
argv[1] = "-ef" ;
argv[2] = (char*)0;
execv( "/bin/ps" , argv);
```

exec는 프로쎄스의 생성과는 전혀 상관없는 프로쎄스대치와 관련된 작업을 수행한다. 하지만 exec를 프로쎄스생성함수인 fork()와 함께 사용하면 전혀 다른 목적을 가진 프로쎄스를 다양하게 실행할수 있는 효과를 얻게 된다. 즉 fork를 통해 자식프로쎄스를 생성시킨 다음에 exec를 리용하여 자식프로쎄스를 다른 프로쎄스로 대치시키는것과 같은 일을 할수 있다.

이를 통하여 부모프로쎄스는 원하는 작업을 수행하는것과는 전혀 성격이 다른 자식프로쎄스를 가질수 있게 된다. 그러면 fork와 exec를 리용하여 이러한 작업을 수행하는 실례프로그람을 보도록 하자.

실례는 fork를 리용하여 자식프로쎄스를 만든 다음 자식프로쎄스를 ps -f로 대치되

도록 만드는 프로그램이다. 이때 부모프로세스는 자식프로세스를 만든 다음 2s가 지나면 완료되도록 해보자.

실례프로그람: ex_exec2.c	
1	#include <stdio.h>
2	
3	/* 자식프로세스가 실행할 exec함수 */
4	void forChild (void)
5	{
6	printf ("<<자식프로세스 ps로 대체>>\n");
7	execl ("/bin/ps", "ps", "-f", (char *) 0);
8	
9	/* 오유발생시에 표시될 통보문 */
10	printf ("exec()실행과정에 오유발생\n");
11	exit (1) ;
12	}
13	
14	/* 프로그람의 주함수 */
15	int main()
16	{
17	/* 자식프로세스의 pid용 변수 */
18	int childPID = 0;
19	
20	/* 실행계수기 초기화 */
21	int parentCount = 0;
22	
23	/* fork함수 실행, 자식프로세스 생성 */
24	childPID = fork() ;
25	
26	/* 자식프로세스이면 forChild()함수 실행 */
27	if (childPID == 0)
28	{
29	printf("<<자식프로세스 생성>>\n");
30	forChild();
31	}
32	/* 부모프로세스이면 내부모듈 실행 */
33	else if (childPID > 0)

```
34          {
35                  printf("<<부모-자식프로쎄스번호: %d>>\n", childPID);
36                  /* 2s간만 실행 */
37                  for(parentCount=0;parentCount<2;parentCount++)
38                  {
39                          /* 실행계수기 출력 */
40                      printf("PARENT PROCESS ? count: %d\n", parentCount);
41
42                          /* 1s간 sleep 수행 */
43                          sleep(1);
44                  }
45          }
46          /* 프로쎄스의 생성에 실패한 경우 */
47          else
48          {
49                  printf("프로쎄스의 생성에 실패했습니다.\n");
50          }
51          return 0;
52  }
```

프로그람을 작성하였으면 콤파일 및 실행을 시켜보자.(그림 2-7) 프로그람을 실행시키면 자식프로쎄스가 생성되고 이어 ps -f가 실행되는것을 볼수 있다. 그리고 부모프로쎄스는 계수를 1s 간격으로 두번 진행한후에 완료되였다는것을 알수 있다.

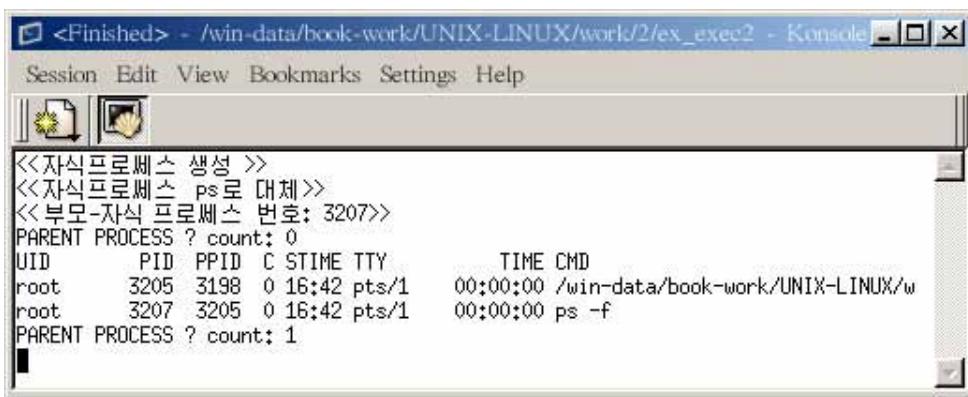



그림 2-7. ex_exec2.c 의 실행결과

2.2.4. Wait

프로쎄스의 동기화(Synchronization)를 위하여 다른 프로쎄스가 끝날 때까지 프로

쎄스를 멈추도록 만들어야 할 때가 있다. 이때 사용하는 함수가 바로 wait()함수이다. 자식프로쎄스를 생성한 이후에 wait()를 호출하면 부모프로쎄스는 자식프로쎄스가 작업을 끝낼 때까지 실행을 중지하게 된다.

따라서 자식프로쎄스가 작업을 끝내면 이어서 부모프로쎄스가 실행을 넘겨받는것과 같은 효과를 얻을수 있다. 만일 자식프로쎄스가 여러개이면 실행중이던 자식프로쎄스중 첫번째로 완료되는 프로쎄스가 나올 때까지 부모프로쎄스는 실행을 중지하고 기다리게 된다.

wait함수의 사용법은 다음과 같다. wait를 실행할 때 자식프로쎄스가 없으면 -1을 되돌려주게 된다.

```
int status;
int result = wait(&status);
```

그러면 wait를 리용한 실례프로그람을 만들어보자. 다음의 실례는 fork()를 리용하여 자식프로쎄스를 생성한 다음 wait()함수를 리용하여 자식프로쎄스가 완료된 다음에 부모프로쎄스가 실행되도록 만든 프로그람이다.

실례프로그람: ex_wait.c	
1	#include <stdio.h>
2	
3	/* 프로그람의 주함수 */
4	int main()
5	{
6	/* 자식프로쎄스의 pid용 변수와 실행계수기 */
7	int childPID = 0;
8	int count = 0;
9	
10	/* fork함수실행, 자식프로쎄스 생성 */
11	childPID = fork();
12	
13	/* 자식프로쎄스 */
14	if (childPID == 0)
15	{
16	printf("<<자식프로쎄스 생성>>\n");
17	/* 2s만 실행 */
18	for(count=0;count<2;count++)
19	{

20	/* 실행계수기 출력 */
21	printf("child process - count:%d\n",count);
22	
23	/* 1s간 sleep수행 */
24	sleep (1);
25	}
26	}
27	/* 부모프로세스 */
28	else if (childPID > 0)
29	{
30	/* wait함수를 실행 */
31	wait((int*)0);
32	
33	printf ("<< 부모-자식프로세스번호:%d>>\n", childPID);
34	/* 2s간만 실행 */
35	for(count=0;count<2;count++)
36	{
37	/* 실행계수기 출력 */
38	printf("parent process - count:%d\n", count);
39	
40	/* 1s간 sleep 수행 */
41	sleep(1);
42	}
43	}
44	/* 프로세스의 생성에 실패한 경우 */
45	else
46	{
47	printf("프로세스의 생성에 실패했습니다.\n");
48	}
49	return 0;
50	}

실례프로그람을 콤파일하고 실행시키면 다음과 같은 결과를 얻게 된다.

실행결과를 통하여 자식프로세스와 부모프로세스가 차례로 실행되고 완료되였다는것을 알수 있다.(그림 2-8)

그림 2-8. ex_wait.c 의 실행결과

Ethernet 란?

콤퓨터망의 자원에 접근하기전에 먼저 다른 사용자의 사용을 감시하는 체계로서 앞에서 설명한 ALOHA 콤퓨터망의 발전된 형태이다. Ether 라는 말은 의학용어로 《에테르》라는 말인데 이 에테르라는것은 옛날 과학자들이 빛의 매질에 대한 연구를 진행하면서 리용하였던 가상적인 물질이다. Ethernet 은 지금까지도 인기가 있는 콤퓨터망인데 초당 최대전송속도는 10Mbps 였으나 최근에 그의 대역폭을 10 배정도 넓힌 Fast Ethernet 가 등장하여 그 약점을 퇴치하고있다.

Ethernet 은 ALOHA 에서 살펴본것과 같이 ALOHA 처럼 동시에 여러개의 마디가 접근을 시도하지 않는다. 왜냐하면 모든 마디들은 콤퓨터망에서 다른 마디의 접근을 감시(또는 청취)하고 충돌이 없을 경우 접근을 시도하기때문이다.

Ethernet 의 경우 충돌이 발생하면 충돌의 원인이 된 마디들에서는 모두 즉시 전송을 중단하고 콤퓨터망을 감시한다. 이러한 Ethernet 의 안정성때문에 최근 근거리 지역망(LAN)에서 많이 쓰이고있다. 보통 망카드를 Ethernet 카드라고도 부르는데 그것은 거의 모든 사무실에 설치되여있는 망이 거의나 다 Ethernet 이기때문이다.

이러한 리유로 하여 Ethernet 을 전문용어로 CSMA-CD 라고 하는데 략자를 풀면 Carrier Sense Multiple Access with Collision Detection 이고 그의 뜻은 《충돌을 감시하여 신호에 다중(복수의 마디)으로 접근가능한 방법》이라고 말할수 있다.

제3절. 프로쎄스 프로그람작성

이 절에서는 다중프로쎄스를 작성하는 과정에 참고로 될만한 비결을 설명하고 이것을 리용하여 실례프로그람들을 작성하는 과정을 통하여 앞에서 설명한 프로쎄스의 리용방법에 대한 지식을 공고히 하도록 한다.

2.3.1. whoami와 ps 활용

프로쎄스를 실행시킬 때 특정한 사용자가 실행시켜야 하는 경우가 있다.

어떤 경우에는 root로 실행을 해야만 작업이 제대로 수행되는 경우도 있다. 이러한 경우에는 프로쎄스를 누가 실행시키는가에 대해 검사할 필요가 있다. 그래서 만일 해당한 사용자가 아니면 경고를 내보내고 프로쎄스를 실행시키지 말아야 한다.

이때 사용할수 있는 쉘지령으로서는 whoami가 있다. whoami를 리용하면 현재 사용자가 누구인가를 알수 있는데 다음 실행권한이 있는 사용자이면 프로그람을 실행시키고 그렇지 않으면 통보문과 함께 완료하면 된다.

체계를 개발하는 과정에는 이미 존재하는 프로쎄스를 완료시켜야 하는 경우가 있다. 또는 프로쎄스가 이미 실행중이라면 새로운 프로쎄스가 실행되지 않도록 만들어야 할 때도 있다. 이러한 때에는 실행중인 프로쎄스를 검사하여야 한다. 다시말하여 프로쎄스가 실행중인가를 검사하고 실행중인 PID를 찾은 다음 이것을 완료시키든지 아니면 새로운 프로쎄스의 실행을 막든지 하는 작업이 필요하게 된다. 이러한 작업을 할수 있는 프로그람을 C언어 등으로 작성하는것은 그렇게 쉬운 일이 아니다. 하지만 쉘프로그람을 리용하면 상당히 쉽게 프로그람을 작성할수 있다. 실례를 들면 ps지령을 리용하여 해당 프로쎄스가 실행중인지 쉽게 알수 있다. 또한 ps를 리용하여 PID를 찾아낸 다음에는 강제완료(kill)지령을 주어 수행중인 프로쎄스를 쉽게 완료시킬수 있다.

또한 뒤면(Back-end)으로 실행되는 데몬을 검사하고 자동실행시키거나 강제완료(kill)시키는 경우에도 이러한 문제를 쉽게 해결할수 있다.

지금까지 설명한 내용에 기초하여 실례프로그람을 만들어보면서 내용을 파악하도록 하자.

먼저 데몬형태로 움직이는 onlyOne이라는 프로쎄스가 root권한으로만 기동한다고 가정하자. 만일 onlyOne이 실행중일 때 누군가가 이것을 또 실행시키려 한다면 더 이상 실행되여서는 안된다. 물론 실행을 시키는 사용자가 root가 아니라도 실행되여서는 안된다.

그러면 onlyOne을 실행시키거나 완료시키는 프로그람인 checkOne이라는 프로그람을 작성해보자.

먼저 onlyOne프로그람을 만들어보자. 이 프로그람은 시험을 위한 무한순환을 돌면서 화면에 통보문을 출력하는 기능만 가지고있으면 된다.

실례프로그람: onlyOne.c	
1	#include <stdio.h>
2	
3	/* 무한순환을 돌며 매초마다 통보문출력 */

4	int main()
5	{
6	while(1)
7	{
8	fprintf(stderr, "ONLYONE PROCESS RUNNING\n");
9	sleep(1);
10	}
11	return 1;
12	}

우의 프로그람을 실행시키면 1s마다 통보문을 출력하면서 계속 실행될것이다. 이제 이 프로그람을 실행 및 완료시키는 checkOne프로그람을 작성해 보자. checkOne프로그람은 실행인수로서 start 또는 stop을 받아서 start이면 onlyOne프로쎄스를 실행시키고 stop이면 onlyOne프로쎄스를 완료하게 된다.

이때 실행시키는 사람이 root인지를 검사하며 이미 실행중인 onlyOne프로쎄스가 있는지 확인하게 된다. onlyOne프로쎄스를 완료해야 하는 경우에는 ps와 kill지령을 가지고있는 쉘프로그람을 리용한다. 그러면 먼저 checkOne프로그람에서 사용할 쉘프로그람부터 작성해보도록 하자.

실례프로그람: checkOne.sh	
1	#!/bin/sh
2	
3	# start 또는 stop 등의 인수의 개수를 검사, 하나만 유효
4	if [$# -eq 1]
5	Then
6	# echo “인수는 한개이며 내용은 <$1>입니다.”
7	Echo “\c”
8	Else
9	# echo “인수가 하나도 없거나 너무 많아 탈퇴합니다.”
10	Echo “INVALIDARG”
11	exit 0
12	Fi
13	
14	# 쉘프로그람을 실행시킨 사용자를 검사, root만 유효
15	user=`whoami`
16	if [$user = root]
17	then

```
18  # echo “사용자는 root입니다.”
19    echo “\c”
20  else
21  # echo “사용자가 root가 아니므로 그냥 탈퇴합니다.”
22    echo “notroot”
23    exit 0
24  fi
25  
26  # 인수가 start인지 stop인지를 검사하고 해당한 모듈을 실행
27  case $1 in
28  start)
29  # 프로세스가 이미 실행중인지 검사하고 없으면 backend로 실행
30    usage=’ps ?a | grep onlyOne | /bin/awk `{print $4}`’
31    if [ “$usage” = “onlyOne” ]
32    then
33  #         echo “onlyOne 프로세스가 이미 실행중입니다.”
34            echo “ALREADY”
35    else
36            onlyOne &
37            echo “RUNNING”
38   % fi ;;
39  stop)
40  # 실행중인 프로세스를 찾아서 강제완료(kill)시킴
41    usage=’ps ?a | grep onlyOne | /bin/awk `{print $4}`’
42    if [ “$usage” = “onlyOne” ]
43    then
44            kill ’ps ?a | grep onlyOne | awk `{print $1}`’ > /dev/null 2>&1
45            echo “stopone”
46    else
47  #         echo “실행중인 프로세스가 없습니다.”
48            echo “anyonep”
49    fi ;;
50  *)
51    echo “unknown arg”
52  esac
53  exit 1
```

우의 프로그람을 보면 먼저 "if 〔 $# -eq 1〕"을 리용하여 인수의 개수가 하나인가를 검사한다. 만일 인수의 개수가 하나도 없거나 두개 이상이면 《INVALIDARG》를 현시하고 탈퇴한다.

그 다음 "user=whoami ; if 〔 $user = root 〕" 문장을 리용하여 사용자가 root인가를 검사한다. 만일 root이외의 사용자이면 《NOTROOT》를 출력하고 프로그람을 끝마친다. 마지막으로 case문을 리용하여 start인 경우와 stop인 경우에 작업해야 하는 모듈을 작성한다.

인수가 start이면 "usage= ps -a | grep onlyOne | /bin/awk '{print $4}' " 문장을 리용하여 onlyOne프로세스가 실행중인가를 확인한다. 만일 실행중이면 《ALREADY》를 출력하고 더 이상 실행하지 않는다. 만일 실행중인 프로세스가 없으면 onlyOne 프로세스를 뒤면(backend)으로 실행한 다음 《RUNNING》이라는 문장을 출력한다.

인수가 stop이면 먼저 실행중인 onlyOne프로세스가 있는지 확인한다. 만일 실행중인 onlyOne 프로세스가 없으면 《ANYONEP》이라는 문장을 출력하고 더 이상 다른 작업을 수행하지 않는다. 만일 실행중인 프로세스가 있으면 "kill ps -a | grep onlyOne | awk '{print $1}' > /dev/null 2>&1" 문장을 리용하여 프로세스를 탈퇴시키고 《STOPONE》을 출력하게 된다.

이번에는 checkOne.sh프로그람을 리용하는 checkOne프로그람을 작성해 보자. checkOne프로그람은 checkOne.sh를 리용하여 onlyOne프로그람을 실행 및 완료하게 된다. 이때 checkOne.sh 프로그람이 되돌려 주는 값을 리용하여 오유여부를 검사하게 된다.

아래에 checkOne.c프로그람의 원천코드를 서술하였다.

실례프로그람: checkOne.c	
1	#include <stdio.h>
2	#include <string.h>
3	
4	/* 프로그람에서 사용하게 될 각본 */
5	static const char * checkone = ". / checkOne . sh";
6	
7	/* function : runCheckOne */
8	/* description : 각본을 실행 */
9	void runCheckOne (char *args, char result [8])
10	{
11	char cmd[32];
12	FILE *fp;
13	
14	/* 인수로 들어온 args를 리용하여 지령완성 */

```
15          sprintf (cmd, "%s %s", checkone, args);
16
17          /* popen으로 작성한 지령문을 실행 */
18          if ((fp = popen(cmd, "rw")) == null)
19          {
20                  fprintf(stderr, "\nrunCheckOne() Failure to open the pipe\n");
21                  strncpy(result, "nullres\0",8);
22                  return;
23          }
24
25          /* 결과를 얻어올 buffer를 초기화 */
26          cmd[0]='F'; cmd[1]='F'; cmd[2]='F'; cmd[3]='F';
27          cmd[4]='F'; cmd[5]='F'; cmd[6]='F'; cmd[7]='\0';
28
29          /* 관에서 각본실행결과를 얻어옴 */
30          fread(cmd, 1, 7, fp);
31          if(!strncmp ("FFFF", cmd, 4))
32          {
33                  pclose(fp);
34                  fprintf(stderr, "\nrunCheckOne() fread failed\n");
35                  strncpy(result, "NULLRES\0", 8);
36                  pclose(fp);
37                  return;
38          }
39
40          /* 인수로 들어온 result에 결과를 보관 */
41          strncpy(result, cmd, 8);
42
43          pclose(fp);
44  }
45
46  /* FUNCTION : int main(int argc, char ** argv) */
47  /* DESCRIPTION : checkOne의 주함수 */
48  int main(int argc, char** argv)
49  {
50          char result[8];
```

51	
52	/* 인수의 수가 2개가 아니면 실행되지 않는다. */
53	if (argc != 2)
54	{
55	fprintf(stderr, "\n\n\n Usage: checkOne[start|stop] \n\n");
56	return 0;
57	}
58	else if(argc == 2)
59	{
60	/* start이면 실행과 관련된 프로세스의 검사를 진행한다. */
61	if(!strncmp("start", argv[1], 5))
62	{
63	/* runCheckOne()함수를 실행하고 결과를 result로 받는다. */
64	runCheckOne(argv[1], result);
65	/* root가 아닐때 오유처리 */
66	if(!strncmp("NOTROOT", result,7))
67	{
68	fprintf(stderr, "\n\n\n You must be root! \n\n");
69	return 0;
70	}
71	/* 이미 실행중일 때 오유처리 */
72	else if(!strncmp("RUNNING", result,7))
73	{
74	fprintf(stderr, "프로쎄스의 실행에 성공!\n");
75	}
76	/* 각본의 실행에 실패한 경우 오유처리 */
77	else if(!strncmp("NULL", result, 4))
78	{
79	fprintf(stderr, "각본파일을 검사하시오.\n");
80	}
81	return 0;
82	}
83	/* stop이면 중지와 관련된 프로쎄스의 검사를 진행한다. */
84	else if(!strncmp("stop",argv[1],4))
85	{
86	/* runCheckOne()함수를 실행하고 결과를 result로 받는다. */

87	runCheckOne (argv[1], result);
88	/* root가 아닐 때 오유처리 */
89	if(!strncmp("NOTROOT", result,7))
90	{
91	fprintf(stderr, "\n\n\n You must be root! \n\n");
92	return 0;
93	}
94	/* 실행중인 프로쎄스가 없을 때 오유처리 */
95	else if(!strncmp ("ANYONE", result,6))
96	{
97	fprintf(stderr, "실행중인 프로쎄스가 없습니다.\n");
98	}
99	/* 중지에 성공한 경우 */
100	else if(!strncmp("STOP", result,4))
101	{
102	fprintf(stderr, "프로쎄스의 완료!!!\n");
103	}
104	/* 각본의 실행에 실패한 경우 오유처리 */
105	else if(!strncmp("NULL", result, 4))
106	{
107	fprintf(stderr, "각본파일을 검사하시오. \n");
108	}
109	return 0;
110	}
111	/* start도 stop도 아닌 인수가 들어온 경우 오유처리 */
112	else fprintf(stderr, "\n\n Usage: checkOne [start|stop] \n");
113	}
114	
115	return 0;
116	}

우의 프로그람을 콤파일하고 실행시켜보자. 아래에 프로그람을 실행한 결과들을 보여주었다. 이를 통하여 프로쎄스의 실행 및 완료, root여부검사 등이 제대로 이루어지고 있는가를 확인할수 있다.

- 인수를 주지 않고 그냥 실행한 경우:

```
% checkOne
Usage: checkOne [start | stop]
```

- root가 아닌 사용자가 실행한 경우:

```
% checkone start
You must be root!
```

- root가 checkOne start를 실행한 경우:

```
% checkOne start
ONLYONE PROCESS RUNNING
checkOne 프로세스 실행 성공!!!
```

- 실행중인데 또다시 start를 실행한 경우:

```
% checkone start
CheCkOne 프로세스가 이미 실행중입니다.
```

- root가 checkOne stop을 실행한 경우:

```
% checkOne stop
프로세스의 완료!!!
```

- 실행이 완료된 상태에서 다시 stop을 실행한 경우:

```
% checkOne stop
실행중인 프로세스가 없습니다.
```

2.3.2. 프로세스 ID 활용

프로세스를 리용하면서 가장 중요하게 활용되는것은 PID이다. 앞의 실례에서 ps를 활용하거나 kill지령을 사용할 때에도 사용되는 열쇠어는 프로세스 ID인 PID였다. 이런 PID를 얻을 때 사용하는 함수는 getpid()이다.

그리고 부모프로세스의 PID를 구하는 함수는 getppid()함수이다. 이 함수들의 간단한 사용방법을 보면 아래와 같다.

```
int processID = getpid() ; /* 자기의 PID를 구하는 경우 */
int parentPID = getppid(); /* 부모의 PID를 구하는 경우 */
```

그러면 getpid()와 getppid()를 리용한 간단한 실례프로그람을 만들어보자.

아래의 실레는 fork()와 getpid(), 그리고 getppid()를 리용하여 자식의 PID와 자기의 PID, 그리고 부모의 PID를 구하여 출력하는 프로그람이다. 여기서는 현재 실행되는 프로세스가 부모인지 자식인지를 검사하는 방법으로 getpid()를 사용하고있다.

실례프로그람: ex_getpid.c	
1	#include <stdio.h>
2	
3	/* 프로그람의 주함수 */
4	int main()
5	{
6	/* 프로쎄스의 PID변수들 */
7	int childPID = 0;
8	int parentPID = 0;
9	
10	/* 자기자신의 PID를 구한다. */
11	parentPID = getpid();
12	
13	/* fork()함수 실행, 자식프로쎄스 생성 */
14	childPID = fork();
15	
16	/* 자식프로쎄스 실행 */
17	if(getpid() != parentPID)
18	{
19	printf("<<자식프로쎄스>>\n");
20	printf("자식 - 자식PID: %d, 부모PID: %d\n", getpid(), getppid());
21	}
22	/* 부모프로쎄스 실행 */
23	else if(getpid() == parentPID)
24	{
25	printf("<<부모프로쎄스>>\n");
26	printf("부모 - 자식PID: %d, 부모PID: %d\n", childPID, parentPID);
27	}
28	return 0;
29	}

우의 프로그람을 콤파일하여 실행시켜보자. 그러면 그림 2-9와 같은 결과가 나올것이다.

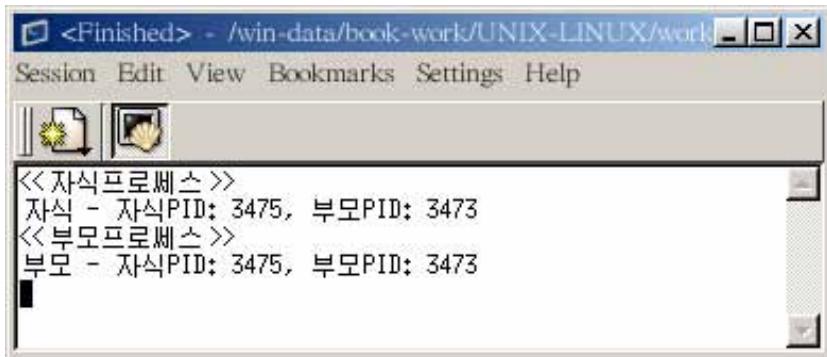



그림 2-9. ex_getpid.c 의 실행결과

getpid()를 리용하여 프로쎄스의 정보중 가장 중요한 값인 PID를 얻을수 있었는데 류사한 함수들을 리용하여 프로쎄스의 기타 정보도 얻을수 있다. 례를 들어 프로쎄스를 실행시킨 사용자의 ID, 그룹의 ID 등도 얻을수 있다. 함수의 이름은 getuid()와 getgid()인데 이것을 리용하면 사용자 ID와 그룹 ID를 각각 구할수 있다.

함수의 사용방법은 다음과 같다.

```
int userID = getuid();
int groupID = getgid();
```

그러면 이러한 함수들을 리용한 간단한 실례프로그람을 만들어보자. 실례프로그람을 콤파일하고 실행시켜보면 프로쎄스의 사용자 ID와 그룹 ID 정보가 화면에 현시되는것을 볼수 있다.

실례프로그람: processinfo.c	
1	#include <stdio.h>
2	
3	/* 프로그람의 주함수 */
4	int main()
5	{
6	/* 프로쎄스의 정보를 보관할 변수들 */
7	int psPID, psUID, psGID;
8	
9	/* 자신의 프로쎄스정보를 얻기 */
10	psPID = getpid();
11	psUID = getuid();
12	psGID = getgid();
13	

```
14        /* 프로세스정보를 화면에 현시한다. */
15        printf("<<프로세스정보>>\n");
16        printf("PID: %d, UID: %d, GID: %d\n", psPID, psUID, psGID);
17        return 0;
18  }
```

우의 실례를 여러 사용자의 ID로 실행해 보면 프로세스의 사용자 ID(user ID)와 그룹 ID(group ID)는 프로세스를 기동한 사용자의 정보와 같다는것을 알수 있다. 실행결과를 그림 2-10에 표시하였다.

getuid(), getgid()함수들과 대조되는 함수로서는 setuid()함수와 setgid()함수가 있다. 이 함수들을 리용하면 프로세스의 사용자 ID와 그룹 ID를 변경할수 있다.

하지만 일반프로세스의 사용자 ID를 특권사용자의 ID로 변경할수는 없다. 오직 그 반대의 경우만이 가능하다. 다시말하여 root가 실행한 프로세스의 경우에는 setuid()를 리용하여 요구하는 사용자의 ID로 바꿀수 있으나 그 반대로는 바꿀수 없다는것이다.

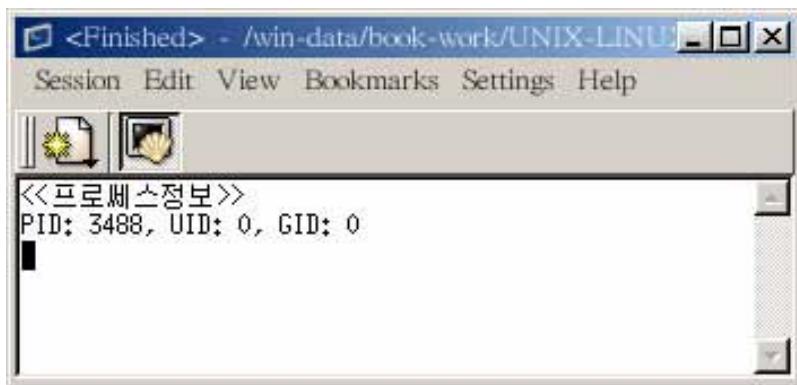



그림 2-10. processinfo.c 의 실행결과

상식

유니코드(UNICODE)란 무엇인가?

세계의 많은 문자들을 통일적인 문자부호로 표현하려고 하는 규격이다.

마이크로쏘프트, 애플콤퓨터회사들을 중심으로 하는 유니코드협회에서 제정하여 ISO(국제표준화기구)에서 국제규격의 일부로서 적용되고있다. 모든 문자를 2byte의 문자부호로서 표현한다.

유니코드를 사용한 응용프로그람은 어떤 언어의 조작체계에서도 리용할수 있게 되여있다. 그러나 2byte로 모든 문자를 수록하기때문에 일본어와 중국어에서는 같은 부호에 서로 차이나는 글자체 등이 할당되는것과 같은 문제도 있다.

제3장

신호

서론

이 장에서는 중요한 프로세스들 사이의 통신기법중의 하나인 신호에 대하여 취급하게 된다.

신호는 정상적인 통보문전송보다도 비정상적인 통보문전송에 가깝기때문에 일반적으로 많이 사용하는 용어인 alarm과 류사하다. 즉 프로세스실행중에 특별한 일이 발생하여 프로세스에 알릴 필요성이 있을 때 핵심부가 통보문을 내보내여 알려주는것이라고 할수 있다.

핵심부로부터 신호를 받으면 프로세스는 이것을 무시하거나 처리할수 있다. 따라서 필요한 신호는 반드시 처리하도록 만들며 그리 중요하지 않은 신호는 완전히 무시하도록 만드는것이 필요하다.

이 장에서는 신호란 무엇이며 신호와 관련된 체계호출함수는 어떤것인가에 대하여 설명한다. 그리고 이와 관련된 프로그람작성방법을 보여준다.

3장은 다음과 같은 절들로 구성되여있다.

목표

1. 신호란 무엇인가
2. 신호처리
3. 신호전송
4. 신호프로그람작성

제1절. 신호란 무엇인가

Linux체계에서 제공하는 프로쎄스들사이의 통신중에서 가장 오래된 기법중의 하나인 신호는 정상적인 자료의 교환을 위한 통신이라기보다는 비정상적인 상태를 서로에게 알려주기 위한 방법으로 많이 사용된다.

체계에 돌발적인 정보를 알리는것을 Notification이나 alarm이라고 부를수 있는데 신호도 이와 류사한것이라고 생각할수 있다. 물론 돌발적인 정황만을 위해서 신호가 존재하는것은 아니지만 신호의 사용법이나 사용목적이 대부분 거기에 알맞게 되여있다.

신호는 새치기가 발생했을 때 이것을 프로쎄스에 알리기 위해서 많이 사용된다. 즉 새치기가 발생했을 때 이를 적절하게 처리하는 루틴을 프로쎄스속에 넣어서 비정상적인 동작을 미연에 방지하거나 정황에 따르는 적절한 조치를 취하기 위해서이다.

신호를 잘 활용하면 새치기를 프로쎄스의 또 다른 결합부로 활용할수 있다. 즉 프로쎄스와의 대화창구로 새치기를 추가할수 있다는것이다. 이것을 리용하면 쉘에서 발생시킨 신호를 리용하여 작업중인 프로쎄스에 다른 명령을 주는것이 가능하게 된다.

신호는 이름을 가지고있는데 이름이 정의되여 있는 곳은 signal.h파일로서 그안에 #define을 리용하여 선언되여있다. signal.h파일속에 많은 신호이름 등이 지정되여 있지만 실제로 개발자들이 사용하는 신호들은 그렇게 많지 않다. 대부분의 신호는 핵심부가 프로쎄스를 관리하거나 조작체계준위의 작업을 수행하는데 사용된다.

signal.h속에 정의된 신호이름들중 중요한 신호들을 소개하면 다음과 같다.

・#define SIGHUP: 우로련결(Hangup)을 위한 신호로서 말단과 체계사이에 통신접속이 끊어졌을 때 말단에 련결된 프로쎄스들에 핵심부가 보내는 신호이다.

・#define SIGINT: 새치기(Interrupt)를 위한 신호로서 사용자가 새치기를 발생시키는 건을 입력했을 때 그와 련결된 프로쎄스에 핵심부가 보내는 신호이다. 이 신호는 프로쎄스가 탈퇴할 때 많이 사용되는 신호이다.

・#define SIGQUIT: 탈퇴(Quit)를 위한 신호로서 사용자가 말단에서 탈퇴건을 누르면 핵심부가 프로쎄스에 SIGQUIT신호를 보낸다.

・#define SIGILL: illegal지령, 즉 비정상적인 지령을 수행할 때 조작체계가 발생시키는 신호이다.

・#define SIGTRAP: Trace Trap를 위한 신호로서 오유수정(debug)을 위해 주로 사용하는 신호이다.

・#define SIGABRT: 탈퇴(Abort)시 발생하는 신호로서 체계가 비정상적으로 탈퇴할 때 해당한 정보를 남기는 지령이다.

・#define SIGIOT: SIGABRT와 류사한 작업을 수행할 때 발생하는 신호이다.

・#define SIGEMT: Emt지령실행시 사용되는 신호이다.

・#define SIGFPE: 류동소수점(Floating point)의 례외사항 즉 고정소수점사용에서 웃자리넘침이나 아래자리넘침이 발생했을 때 사용되는 신호이다.

・#define SIGKILL: 한프로쎄스가 다른 프로쎄스를 강제완료(Kill)시키기 위해 사용하는 신호이다.

・#define SIGBUS: 모선(BUS)오유가 발생했을 때 사용하는 신호이다.

・#define SIGSEGV: 기억기토막 등이 깨졌을 때 사용하는 신호이다.

・#define SIGSYS: 체계호출을 할 때 잘못된 인수를 사용하면 사용하는 신호이다.

・#define SIGPIPE: 관(Pipe)에서 사용하는 신호로서 아무도 리용하지 않는 관으로 자료를 출력할 때 사용하는 신호이다.

・#define SIGTERM: 강제완료(Kill)에 의해 프로쎄스가 완료될 때 사용되는 신호이다.

이밖에도 많은 신호가 존재한다.

대부분의 신호들이 핵심부가 프로쎄스에 보내는것이라고 했는데 프로쎄스안에서 신호들을 처리하지 않는다면 대부분의 신호들은 그냥 무시된다. 다시말하여 아무리 핵심부가 신호를 보내도 프로쎄스가 받아서 쓰지 않으면 의미가 없는 통보문이 될수 있다.

그렇다고 프로쎄스가 모든 종류의 신호을 무시할수 있는것은 아니다. 실례로 SIGKILL신호인 경우에는 프로쎄스가 완료되기때문에 무시하는것과는 상관없이 프로쎄스에 영향을 미치게 된다. 프로쎄스가 신호를 무시하는 경우에는 핵심부에 처리를 맡기는 결과를 가져온다.

만일 프로쎄스가 신호를 무시하지 않고 관리를 한다면 신호가 발생했을 때 이를 어떻게 처리하겠는가를 프로쎄스가 결정할수 있다.

하지만 신호의 관리가 그렇게 쉬운것은 아니다. 서로 다른 종류의 신호가 동시에 프로쎄스에 들어올수도 있고 동일한 신호가 계속해서 들어올수도 있다. 따라서 모든 종류의 신호을 처리한다는것은 아주 힘든 작업이기때문에 해당 프로쎄스가 특별히 중요하지 않은 신호들은 신경을 쓰지 않고 중요하다고 생각되는 신호에 대해서는 처리를 진행하도록 하는것이 필요하다.

신호는 프로쎄스와 밀접한 관련이 있기때문에 신호정보는 프로쎄스표(process table)에서도 관리할수 있다. 이 방법으로는 프로쎄스표안에 설정된 내용에 따라 신호처리를 결정하게 되고 또한 신호를 처리한 내용이 프로쎄스정보에 보관되기도 한다.

신호가 프로쎄스들사이의 통신이라고는 하지만 신호는 사용자에 따라 제한이 있다. 다시말하여 일반사용자는 원하는 모든 프로쎄스에 신호를 보낼수 없다는것이다. 물론 특권사용자는 원하는 프로쎄스에 신호를 발생시킬수 있다.

프로쎄스는 신호를 블로크화시킬수 있다. 블로크화된 신호는 블로크가 해제될 때까지 기다려야 하는데 해제되면 프로쎄스에 전달된다. 이때 해제되려면 블로크화되지 않은 신호가 발생하거나 체계호출작업이 이루어져야 한다. 대부분의 프로쎄스가 체계호출을 매번 사용하기때문에 블로크화된 신호가 무한정 기다리게 되는것은 아니다.

신호를 핵심부가 아닌 프로쎄스에서 처리하려고 하면 sigaction속에 신호의 처리를 위한 조종기가 등록되여야 한다. 만일 특별한 신호조종기가 등록되지 않으면 핵심부가 기본적인 처리를 담당하게 된다. 조종기가 sigaction안에 등록이 되면 신호가 발생할 때 매번 호출이 된다. 조종기가 호출될 때에는 마스크를 리용하여 무시할 신호에 대해서는 핵심부에 처리를 맡기게 된다.

지금까지 신호에 대한 개괄적인 내용에 대해 설명하였다. 이제부터 신호의 검출 및 처리 또는 발생시킬 때 사용하는 체계호출(함수)에 대해서 보도록 하자.

제2절. 신호처리

이 절에서는 신호처리를 위해 제공되고있는 체계호출함수들에 대해서 고찰한다. 먼저 핵심부로부터 들어오는 신호를 처리하기 위해 사용되는 Signal체계호출함수에 대해 보도록 하자.

Signal

signal()함수는 이름 그대로 신호를 처리하기 위한 함수이다. 핵심부가 발생시킨 신호 등을 프로쎄스가 받았을 때 이를 처리할 함수를 지정하는 기능을 가지고있다. signal()함수의 간단한 사용실례를 보면 다음과 같다.

```
int sigKind;
int function();
signal(sigKind, function);
```

첫번째 인수인 sigKind는 처리하려고 하는 신호를 의미한다. 이때 프로쎄스가 완료되여야 하는 SIGKILL외에는 sigKind로 지정하여 사용할수 있다. 두번째 인수로 사용되는 함수의 이름은 지정한 신호가 발생했을 때 처리를 담당할 함수를 지정한것이다.

이때 두번째 인수로 함수이름밖에 SIG_IGN이나 SIG_DFL기호를 사용할수 있다. 즉 다음과 같이 사용할수도 있다.

```
signal(sigKind, SIG_IGN); 또는 signal(sigKind, SIF_DFL);
```

SIG_IGN기호는 해당 신호를 무시하라고 지정하는 기호이며 SIG_DFL기호는 표준신호조종기에 처리를 맡긴다는것을 의미한다. 그러면 signal()함수를 리용한 간단한 실례프로그람을 먼저 보도록 하자.

다음의 실례는 프로쎄스가 실행중에 발생한 SIGINT신호를 처리하는 과정을 보여준다. 이를 위해 signal()함수를 리용하여 SIGINT신호와 이를 처리할 조종기를 등록해준다.

실례프로그람: ex_signal.c

```
1   #include <signal.h>
2   
3   /* SIGINT신호를 처리할 조종기 */
4   int sigintHandler()
5   {
6           /* 필요한 작업을 처리한후 프로그람을 완료 */
7           printf("\n\nSIGINT 조종기호출 \n");
8           printf("\n<<<작업완료>>>\n");
9           sleep(1);
10          printf("\n\nStop all run process\n");
11          printf("All open file closed\n\n\n");
12          exit(1);
13  }
14  
15  /* 프로그람의 main 함수 */
16  int main()
17  {
18          int result, step = 0;
19          /* SIGINT조종기를 등록, signal() 함수사용 */
20          printf("SIGINT조종기 설정 \n\n");
21          result = signal(SIGINT, sigintHandler);
22  
23          /* 프로그람 실행, 무한순환 */
24          printf("\n<<<Main프로쎄스실행>>>\n");
25          printf("File open 실행\n");
26          while(1)
27          {
28                  step++;
29                  printf("%d번째 작업수행\n", step);
30                  sleep(1);
31          }
32          Return 1;
33  }
```

우와 같이 프로그람을 작성하고 실행을 시켜보자. 프로그람이 실행되면 SIGINT 신호를 발생시키기 위해 프로쎄스가 실행중인 쉘에서 CTRL+C건을 입력한다. 신호를

처리하지 않는 일반적인 프로그람은 CTRL+C건를 입력하면 핵심부가 프로세스를 완료하게 된다.

일반적인 경우 신호가 발생했을 때 프로세스는 처리하던 체계호출을 수행한 다음 핵심부로부터 신호를 받게 된다. 따라서 처리중인 체계호출은 신호의 영향을 받지 않는다. 하지만 모든 체계호출이 다 그런것은 아니다. 처리속도가 느린 장치에서 체계를 호출하는 경우에는 신호로 하여 작업도중에 체계호출이 중단되면서 프로세스가 완료된다.

만일 신호처리를 제대로 해두지 않으면 파일이나 장치를 사용하던 도중에 작업과정이 없어지는 현상이 발생할수 있기때문에 신호처리를 통해 이에 대처하는것이 좋다. 실례로 신호조종기를 리용하여 열린 파일을 차례로 닫는다든지 필요한 자료를 저장해두는것이 좋다.

프로세스가 실행중에 signal()함수를 만나면 해당 신호에 대한 기본조종기(처리함수)를 지정된 조종기로 변경하게 된다. 따라서 조종기를 필요에 따라 변경해야 한다면 해당 루틴속에서 signal()을 리용하여 조종기를 재지정하면 된다.(그림 3-1) 이후에 발생하는 신호는 재지정된 조종기에 의해 처리가 된다.

이번에는 signal체계호출함수를 리용하여 종류가 다른 신호에 대한 조종기를 등록하고 이를 처리하는 프로그람을 실례로 보도록 하자. 다양한 신호를 처리하기 위해 이러한 작업은 필수적으로 제기되는데 하나의 신호조종기를 등록하는것과는 작업상에서 차이가 크다.

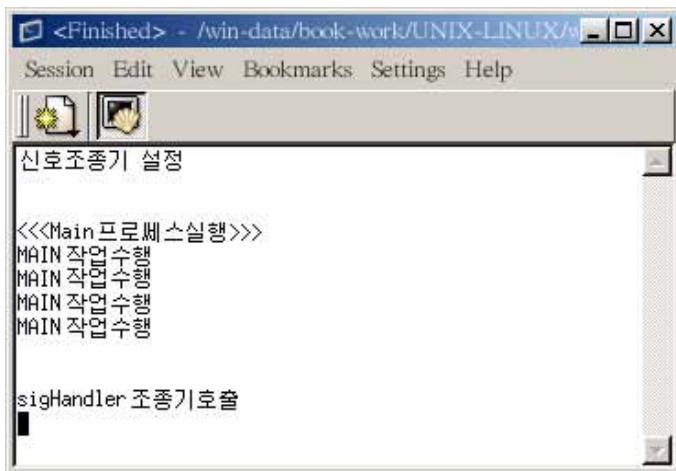



그림 3-1. ex_signal.c의 실행결과

아래의 실례는 SIGINT신호와 SIGQUIT신호를 처리하는 례제이다. 두 종류의 신호를 모두 동일한 조종기인 stopHandler()함수가 처리하도록 등록을 시켜주고있다.

실례프로그람: ex_sigquit.c	
1	
2	#include <signal.h>
3	
4	/* 작업완료신호를 처리할 조종기 */

```
5  int stopHandler()
6  {
7          /* 필요한 작업을 처리한후 프로그람을 완료 */
8          printf("\n\n신호조종기 호출\n");
9          printf("\n<<<작업완료>>>\n");
10         sleep(1);
11         printf("\n\nStop all run process\n");
12         printf("All open file closed\n\n\n");
13         exit(1);
14 }
15
16 /* 프로그람의 main 함수 */
17 int main()
18 {
19         int result, step = 0;
20
21         /* 신호조종기를 등록, signal()함수사용 */
22         printf("SIGINT, SIGQUIT 조종기 설정\n\n");
23         result = signal(SIGINT, stopHandler);
24         result = signal(SIGQUIT, stopHandler);
25
26         /* 프로그람 실행, 무한순환 */
27         printf("\n<<<Main프로쎄스실행>>>\n");
28         printf("File open실행\n");
29         while(1)
30         {
31                 step++;
32                 printf("%d번째 작업수행\n", step);
33                 sleep(1);
34         }
35         return 1;
36 }
```

SIGQUIT신호는 체계의 완료를 위해 사용하는 건인 'CTRL+\'를 리용하여 발생시킬수 있다. 프로그람작성이 끝났으면 실행을 시켜보자. 프로그람의 실행을 통하여 SIGINT신호와 SIGQUIT신호를 처리하였다는것을 확인할수 있다.(그림 3-2)

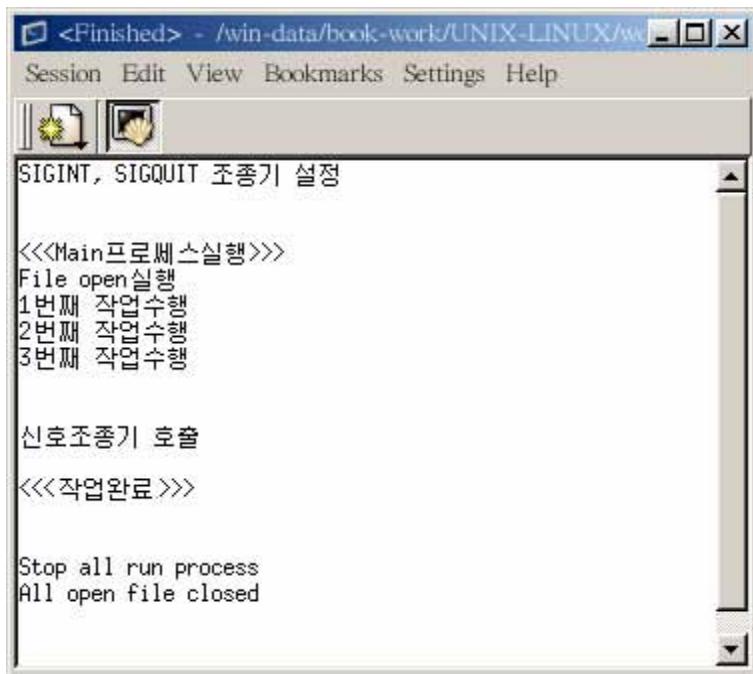



그림 3-2. ex_sigquit.c 의 실행결과

상식

세계를 대상으로 하는 WWW

콤퓨터를 잘 모르는 사람들도 신문이나 방송을 통하여 WWW 라는 이야기는 들었을것이다. WWW 는 World Wide Web 의 략자로서 유럽의 한 연구사가 개발한 **하이퍼본문형식의 분산정보체계**이다. WWW 는 오늘날 인터네트상에서 가장 인기를 끌고있으며 통신에서의 다매체화실현에 중추를 담당한 체계라고 볼수 있다.

하이퍼본문형식, 화상, 음성, 동화상 등을 조합하여 정보를 제공할수 있는데 하이퍼본문의 우점은 어떤 정보에서 필요한 부분을 선택하면 그에 대한 자세한 정보를 얻을수 있다는것이며 그와 같은 정보의 련결이 세계적인 범위에서 그물처럼 펼쳐져있다는것이다.

WWW 의 특징은 전세계의 WWW 봉사기들이 서로 련결되여 통신에서의 다매체화를 실현할수 있게 해준다는데 있다.

제3절. 신호전송

대부분의 신호가 핵심부에 의해 발생되고 핵심부에 의해 처리되지만 사용자가 프로세스에 신호를 전송할수도 있다. 이를 위해 사용하는 체계호출함수로서 널리 리용되는것이 kill()함수이다.

3.3.1. Kill

kill()함수를 리용하여 해당 프로세스의 PID와 신호를 인수로 넣어주면 원하는 신호가 전송된다. kill()함수의 간단한 사용형식을 보면 다음과 같다.

```
int processID, sigKind;
kill(processID, sigKind);
```

실례로 신호를 전송하려는 프로세스의 PID가 815번이라고 할 때 이 프로세스에 프로세스완료를 위한 SIGTERM신호를 보내려면 다음과 같이 하면 된다.

```
kill(815, SIGTERM)
```

그러면 kill()함수를 리용한 실례프로그람을 작성하여 보자. 아래의 실례에서는 rcvSignal 프로세스가 먼저 실행하면서 자기의 PID를 화면에 출력한 다음 작업을 수행하게 된다. 그다음 sendSignal프로세스가 rcvSignal프로세스의 PID를 리용하여 SIGTERM신호를 전송하게 된다.

SIGTERM신호를 받은 rcvSignal프로세스는 stopHandler를 통하여 프로세스가 안전하게 완료되도록 작업을 처리한다.

rcvSignal프로그람의 원천코드는 아래와 같다.

실례프로그람: rcvSignal.c	
1	#include <signal.h>
2	
3	/* SIGTERM 신호를 처리할 조종기 */
4	int stopHandler()
5	{
6	/* 필요한 작업을 처리한후 프로그람을 완료 */
7	printf("\n\n신호조종기호출\n");
8	sleep(1);
9	printf("\n<<<작업완료>>>\n\n");

10	exit(1);
11	}
12	
13	/* 프로그람의 main 함수 */
14	int main()
15	{
16	int result, step = 0;
17	
18	/* 신호조종기를 등록, signal()함수의 사용 */
19	printf("SIGINT조종기설정\n\n");
20	result = signal(SIGTERM, stopHandler);
21	
22	/* 프로그람 실행, 무한순환, 자신의 PID를 출력 */
23	printf("\n<<<Main프로세스실행, PID:%d>>>\n", getpid());
24	printf("File open 실행\n");
25	while(1)
26	{
27	step++;
28	printf("%d번째 작업수행\n", step);
29	sleep(1);
30	}
31	return 1;
32	}

이번에는 sendSignal프로그람의 원천코드를 보자. sendSignal프로그람은 PID를 인수로 받아 실행된다. 만일 PID를 생략하고 프로그람을 실행시키면 아무런 작업도 수행하지 않고 완료된다.

그러나 PID와 함께 sendSignal프로그람을 실행시키면 해당한 PID에 SIGTERM신호를 전송하게 된다. 원천코드는 아래와 같다.

실례프로그람: sendSignal.c	
1	#include <signal.h>
2	
3	/* main함수 */
4	int main(int argc, char* argv[])

5	{
6	int processID;
7	
8	/* 프로세스의 ID를 파라메터로 받는다. */
9	if (argc != 2)
10	{
11	printf("\n\n Usage: sendSignal processID\n\n\n");
12	exit(0);
13	}
14	
15	/* 인수로 받은 PID를 정수로 변환한 다음, kill()함수를 호출 */
16	processID = atoi(argv[1]);
17	kill(processID, SIGTERM);
18	
19	exit(1);
20	}

우의 프로그램들을 작성한데 기초하여 매개 프로쎄스를 차례로 기동시켜보자. 먼저 rcvSignal을 실행시키면 그림 3-3과 같이 PID가 화면에 현시되면서 작업을 수행하게 된다.

그림 3-3. rcvSignal.c의 실행결과

이번에는 SendSignal프로그람을 다른 쉘상에서 실행시켜보자. 이때 rcvsignal프로세스의 PID를 인수로 사용하여 다음과 같이 실행시킨다.

```
%SendSignal 3542
```

그러면 앞서 실행시켰던 rcvSignal프로세스는 실행하고있는 쉘화면에 다음과 같은 내용(그림 3-4)을 출력하면서 프로세스를 완료하게 된다.

지금까지 kill()함수를 리용하여 다른 프로세스에 신호를 전송하는 실례를 보았는데 kill()함수를 리용하여 자기 자신에게 신호를 보낼수도 있다. 이를 위해서는 인수로 사용하는 PID를 리용하면 된다.

례를 들어 kill(0, SIGTERM)을 실행시키면 kill()함수를 호출한 프로세스와 같은 그룹에 속해 있는 프로세스들에 SIGTERM신호가 전송된다. 그리고 kill(-1, SIGTERM)을 실행하면 kill()함수를 호출한 프로세스와 같은 사용자가 사용중인 프로세스들에 SIGTERM신호를 전송한다.

알아둡시다.

만일 kill(-1, SIGTERM)을 실행한 프로세스의 사용자가 root이면 대부분의 프로세스들에 SIGTERM 신호가 전송된다.

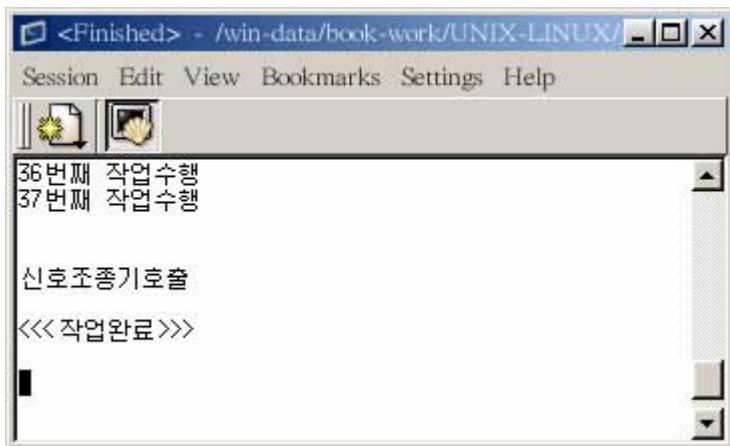



그림 3-4. sendSignal.c의 실행결과

3.3.2. Alarm

alarm()체계호출함수는 우리가 흔히 알고있는 alarm과 같은 동작을 수행한다. 다시 말하여 시계에 경보시간을 맞추어놓으면 시간이 되였을 때 경보가 울리는것과 같다고 볼수 있다. 이처럼 프로세스안에서도 박자계수기가 존재하는데 alarm()함수를 리용하여 박자계수기를 설정해두면 시간이 다 되였을 때에는 SIGALRM(신호시간경보)신호가 발생

하게 된다.

alarm의 설정은 alarm()함수를 호출하여 원하는 시간(초)를 지정하면 된다. 실례로 다음과 같이 alarm을 호출하면 3s후에 SIGALRM신호가 발생한다.

```
alarm (3);
```

만일 alarm신호가 발생하지 않도록 하려면 아래와 같이 값을 0으로 설정하면 된다.

```
alarm (0);
```

그러므로 지정된 시간안에 원하는 작업이 수행되지 않아서 신호를 발생시키려고 할 때에는 alarm()을 리용하면 된다. 그리고 지정된 시간안에 작업이 수행되였으면 0으로 설정하여 alarm을 해제하면 된다. 이것은 우리가 경보시계를 사용하는것과 같다고 말할수 있다. 즉 경보가 울리기전에 잠에서 깨여나 시계를 끄면 되지만 깨여나지 못하여 그냥 놔두면 계속 경보음이 울리는것과 마찬가지로 제때에 작업을 수행하지 못하면 원하지 않는 alarm신호가 발생하게 된다.

그러면 alarm()을 리용한 실례를 보도록 하자. 아래의 프로그람은 scanf()를 리용하여 사용자로부터 수자를 입력받는 프로그람이다. 이때 만일 3s내에 사용자가 입력을 하지 않으면 alarm신호를 리용하여 시간이 다 되였음을 알리고 표준값인 -1로 설정하게 된다. 만일 사용자가 3s이내에 수자를 입력하면 alarm은 해제되며 입력한 수자로 설정된다.

실례프로그람: ex_alarm.c	
1	#include <signal.h>
2	
3	/* 사용자로부터 값을 입력받을 변수 */
4	int defaultVal;
5	
6	/* alarm에 의해 호출될 함수 */
7	int alarmHandler()
8	{
9	/* 입력시간이 다 되였음을 알리고 -1로 설정 */
10	printf("\n시간이 다 되였습니다.\n");
11	defaultVal = -1;
12	return 1;
13	}
14	
15	/* ex_alarm의 main함수 */

16	int main()
17	{
18	/* SIGALARM신호를 처리할 함수를 등록 */
19	signal(SIGALRM, alarmHandler);
20	/* alarm을 리용하여 입력시간설정 */
21	alarm(3);
22	/* 사용자로부터 값을 입력 받음 */
23	printf("3s안에 입력하시오. DEFAULT VALUE:");
24	scanf("%d", &defaultVal);
25	
26	/* alarm에 지정한 초과시간 해제 */
27	alarm(0);
28	
29	/* 마지막으로 입력된 값을 화면에 출력 */
30	printf("DEFAULT VALUE: %d\n", defaultVal);
31	return 1;
32	}

우의 프로그람을 실행해 보도록 하자. 그림 3-5에서는 프로그람을 실행시킨 다음 3s에 값을 입력한 결과를 보여주고있다.

만일 3s안에 값을 입력하지 않으면 아래와 같이 alarm신호에 의해 -1로 설정된다.(그림 3-6)

우의 실례에서 본것과 같이 alarm()을 호출하면 내부적으로 박자계수기가 동작하게 된다. 그리고 프로세스는 alarm()의 다음 행을 계속해서 실행하게 된다. 만일 alarm()을 실행한후에 alarm신호가 발생할 때까지 작업을 중지하고 싶을 때에는 pause()체계호출함수를 사용하면 된다. pause()함수는 시간에 따르는 신호가 발생할 때까지 프로세스를 중지시키는데 유용하게 사용할수 있다. 례를 들어 우의 실례에서 pause()를 다음과 같이 사용했다고 보자.

```
/* alarm 을 리용하여 입력시간 설정 */
alarm(3);
pause();
```

이렇게 되면 alarm(3)이 실행되자마자 pause()가 실행되여 3s간 alarm신호를 기다리게 된다. 3s가 지나면 alarm신호가 발생하고 alarm조종기가 동작하게 된다. alarm조종기의 동작이 끝나면 pause()의 다음 행을 실행하게 된다.

alarm()은 alarm을 실행한 프로세스에는 작용을 하지만 자식프로세스에는 영향을 미치지 않는다. 즉 fork()로 프로세스를 만들었을 때 부모프로세스에만 alarm()이 작용하게 된다.

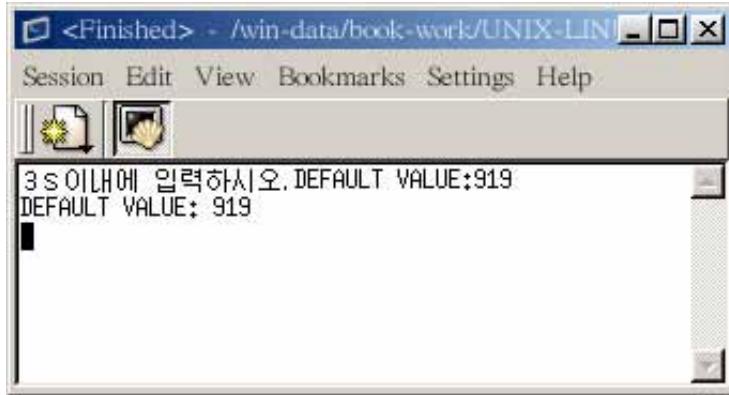



그림 3-5. ex_alarm.c 의 실행결과(1)

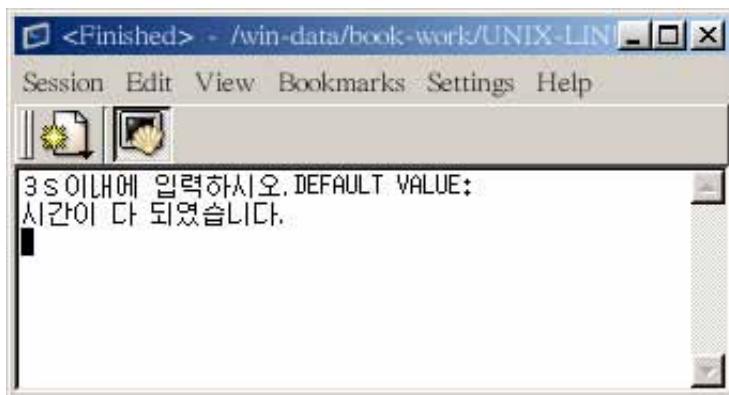



그림 3-6. ex_alarm.c 의 실행결과(2)

3.3.3. Raise

프로세스안에서 신호를 발생시키는 또 다른 체계호출함수로는 raise()가 있다. raise함수를 발생시키려고 하는 신호와 함께 호출해주면 된다. 례를 들어 SIGTERM신호를 내부에 발생시키려면 다음과 같이 하면 된다.

```
raise ( SIGTERM );
```

프로세스가 동작하는 과정에 잘못된 작업이 진행되어 완료를 해야 한다면 raise를 리용하여 신호를 발생시키고 해당 신호조종기가 프로세스를 안전하게 완료시키면 된다. 이것은 외부의 신호를 처리하기 위해 만들어 둔 신호조종기를 내부에서 호출하여 사용하려

고 할 때 많은 도움이 된다.

그러면 raise를 리용한 실례를 보자. 아래의 프로그람은 사용자가 수자를 인수로 사용하여 프로그람을 실행하면 해당한 시간(초)만큼 작업을 수행한 후 raise를 통해 프로그람이 완료되게 한다.

실례프로그람: ex_raise.c

```
1   #include <signal.h>
2   
3   /* raise가 발생시킨 sigterm신호를 처리 */
4   int commonStop()
5   {
6           /* 필요한 작업을 처리한후 프로그람을 완료 */
7           printf("\n\n신호조종기호출\n");
8           sleep(1);
9           printf("\n<<<작업을 완료>>>\n\n");
10          exit(1) ;
11  }
12  
13  /* 프로그람의 main 함수 */
14  int main(int argc, char* argv[])
15  {
16          int secs, steps=1;
17          /* 파라메터의 개수를 검사 */
18          if(argc == 2)
19          {
20                  /* 파라메터로 받은 시간(초)을 정수로 변환 */
21                  secs = atoi(argv[1]);
22                  /* sigterm에 대한 조종기 등록 */
23                  signal(SIGTERM, commonStop);
24  
25                  /* 파라메터로 받은 시간(초)만큼 작업을 수행 */
26                  while(steps <= secs)
27                          {
28                                  sleep(1);
29                                  printf("%d번째 작업을 처리중\n",steps);
30                                  steps++;
```

31	}
32	/* 내부프로쎄스에 sigterm신호발생 */
33	raise(SIGTERM);
34	}
35	/* 파라메터의 개수가 맞지 않을때 */
36	else
37	{
38	printf("\n\n Usage: ex_raise no_of_secs\n\n\n");
39	}
40	return 1;
41	}

우와 같이 프로그람을 작성하였으면 콤파일을 하고 프로그람을 실행해보자. 아래와 같이 3을 인수로 사용하여 프로그람을 실행하면 3s후에 raise를 통해 신호가 발생하고 신호조종기가 프로쎄스를 완료시키게 된다.(그림 3-7)

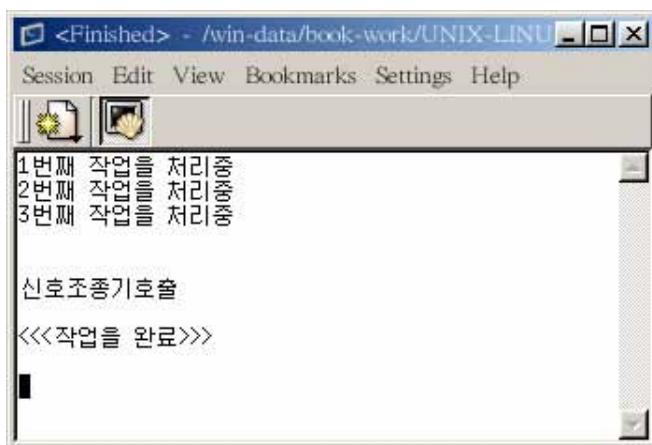



그림 3-7. ex_raise.c의 실행결과

상식

ICMP란 무엇인가?

ICMP는 Internet Control Message Protocol, 즉 인터네트 조종 통보문 규약의 략자이다. 주콤퓨터(Host)봉사기와 인터네트관문(Gateway)사이에서 통보문을 조종하고 오유를 알려주는 규약으로서 RFC 792에 정의되여있다. 주로 경로기가 보내는 IP 계층상의 콤퓨터망정보나 오유상태를 망관리자에게 통보문형태로 전달해주는 기능을 수행한다.

제4절. 신호프로그람의 작성

이 절에서는 앞에서 설명하지 않은 함수와 프로그람작성기법을 리용하여 프로그람을 작성하는 방법에 대해 취급한다. 물론 우에서 취급한 함수들과 기법들을 리용한다. 그리고 쉘에서 신호를 처리하는 방법에 대해서도 설명한다.

3.4.1. 쉘프로그람과 신호

쉘프로그람도 실행도중에 신호가 발생하여 비정상적으로 탈퇴할수 있다. 비정상적인 탈퇴가 진행되면 실행중인 자료가 없어지거나 림시파일을 삭제하지 못하는 등 작업과정이나 그 마무리를 제대로 할수 없다.

이것을 방지하기 위해서는 쉘프로그람안에서도 신호를 처리해주어야 한다. 이러한 작업을 수행하기 위해 사용되는 지령이 trap이다. trap를 리용하여 실행할 지령과 처리하려고 하는 신호를 지정해주면 된다. 만일 실행할 지령을 지정하지 않고 처리할 신호를 지정하면 해당 신호는 무시되는것으로 된다.

trap지령의 간단한 사용방법을 보면 다음과 같다. 다음의 경우는 INT신호(SIGINT와 같음)가 발생하면 지령1과 지령2가 자동적으로 실행되도록 지정한것이다.

```
trap "지령 1; 지령 2" INT
```

그러면 trap를 리용하여 신호를 처리하는 쉘프로그람을 작성한 다음 이것을 실행해 보도록 하자. 아래의 useTrap.sh프로그람은 find지령을 통해 a라는 문자를 가진 파일을 tmp.txt에 보관하도록 만들어주는 프로그람이다.

아래의 실레프로그람은 find지령을 수행하는 도중에 새치기가 발생하여 신호가 검출되면 tmp.txt파일을 삭제하면서 프로세스를 탈퇴하는 프로그람으로서 원천코드는 다음과 같다.

실례프로그람: useTrap.sh	
1	#!/bin/sh
2	trap "echo Caught SIGINT...Removing tmp.txt; rm tmp.txt; exit" INT
3	find . -name "*" -exec grep -l a {} \; > tmp.txt

코드작성이 끝났으면 프로그람을 실행시켜보자. 프로그람을 실행할 때에는 등록부가 많은 곳에서 실행하도록 하여야 한다. 그렇지 않으면 Ctrl+C를 누르기전에 프로그람의 실행이 완료될수 있다. 신호를 발생시킨후 해당 등록부를 보면 tmp.txt파일이 존재하지 않는것을 볼수 있다. 만일 신호처리를 하지 않으면 tmp.txt파일은 지워지지 않는다.

3.4.2. 프로그람의 복귀

신호를 처리할 조종기를 등록한 상태에서 프로세스가 실행되였다고 가정하자. 프로세스가 실행되는 도중에 해당한 신호가 발생하여 신호가 처리되였는데 만일 이때 프로세

스를 초기화한후 다시 처음부터 실행시키고 싶다거나 혹은 처음부터 실행시키는 경우가 아니더라도 신호를 처리한 다음 원하는 곳으로 가서 계속 실행되여야 할 때도 있을것이다.

례를 들어 주차림표와 부차림표가 있는데 부차림표의 실행에서 문제가 발생했거나 사용자가 Ctrl+C를 눌렀을 때 빨리 주차림표로 복귀하도록 만들어야 할 때가 있다. 이러한 작업이 가능하려면 복귀하려고 하는 곳의 위치와 상태가 특정한 기억기에 보관되여야 하며 그곳으로 복귀할수 있는 방법이 제공되여야 한다. Linux의 체계호출함수로서 이것을 가능하게 만드는 함수가 setjmp()와 longjmp()이다. setjmp()는 복귀하려고 하는 곳의 위치를 지정하는 함수로서 이때 체계의 탄창에 위치와 상태정보가 보관된다. 그리고 longjmp()를 호출하면 setjmp()의 위치로 프로그람의 지적자가 옮겨지며 계속 작업을 수행하게 된다. setjmp()와 longjmp()의 간단한 사용방법을 보면 다음과 같다.

```
jmp_buf jmpPos; /* 뛰여넘기할 위치를 가지는 변수선언 */
setjmp(jmpPos);
longjmp(jmpPos, 0);
```

만일 longjmp에서 넘겨주는 값을 setjmp가 받도록 하려면 다음과 같이 한다.

```
int jmpcount = 0;
jmpCount++;
longjmp(jmpPos, jmpCount);/*longjmp의 인수에 jmpCount를 입력*/
result = setjmp(jmpPos);/*result에 jmpCount 값이 들어옴*/
```

이 기능을 리용하면 longjmp로부터 원하는 값을 전달받을수 있을뿐아니라 longjmp가 제대로 수행되였는가 하는것도 검사할수 있다. 그러면 setjmp와 longjmp를 활용한 실례프로그람을 만들어보자.

아래의 실례프로그람은 프로쎄스의 시작지점에 setjmp를 설정한후 신호가 발생하면 작업을 실행할 신호조종기안에서 longjmp가 실행되도록 만든 프로그람이다. 따라서 신호처리와 프로쎄스의 시작이 반복되는 결과를 가져온다. 이때 longjmp가 세번 실행되면 프로그람은 탈퇴하게 된다. 이를 위해 이행회수를 계수하여 몇번 실행되였는지 검사한다.

실례프로그람: ex_jmp.c	
1	#include <signal.h>
2	#include <setjmp.h>
3	
4	/* jmp할 위치를 가지는 변수선언 */
5	jmp_buf jmpPos;
6	/* jmp시 증가하는 계수기 */

7	int jmpCount;
8	
9	/* jmp를 수행하는 함수 */
10	int runJump()
11	{
12	/* jmp계수기 증가, 계수기를 리용하여 완료시점 검사 */
13	jmpCount++;
14	if(jmpCount < 3)
15	{
16	printf("\n아직 완료할수 없습니다.\n");
17	}
18	else
19	{
20	printf("\n완료모듈로 이동\n");
21	}
22	
23	/* 이 시점에서 발생하는 sigint신호는 무시 */
24	signal(SIGINT, SIG_IGN);
25	
26	/* 지정된 위치로 복귀, jmp의 계수기를 파라메터로 활용 */
27	longjmp(jmpPos, jmpCount);
28	}
29	
30	/* main()함수 */
31	int main()
32	{
33	/* setjmp로부터 값을 받을 변수의 초기화 */
34	int result = 0;
35	/* jmp계수기 초기화 */
36	jmpCount = 0;
37	
38	/* setjmp실행, longjmp가 실행되면 result값을 변화 */
39	result = setjmp(jmpPos);
40	
41	/* longjmp()가 세번 실행되였으면 system을 완료 */
42	if(result >= 1)

43	{
44	printf("\n system완료중...\n");
45	sleep(2);
46	exit(1);
47	}
48	
49	/* 신호조종기 설정 */
50	signal(SIGINT, runJump);
51	
52	/* 주프로그람 실행, result값을 리용하여 통보문작성 */
53	while(1)
54	{
55	sleep(1);
56	printf("%d번째 단계로 프로그람 실행중...\n",result);
57	}
58	return 1;
59	}

그러면 프로그람을 콤파일하고 실행시켜보자. 프로그람을 실행시킨 다음 Ctrl+C건을 리용하여 신호를 세번 전송시키도록 한다. 신호가 3번 실행되면 프로세스는 탈퇴한다. 그림 3-8에 ex_jmp.c의 실행결과를 표시하였다.

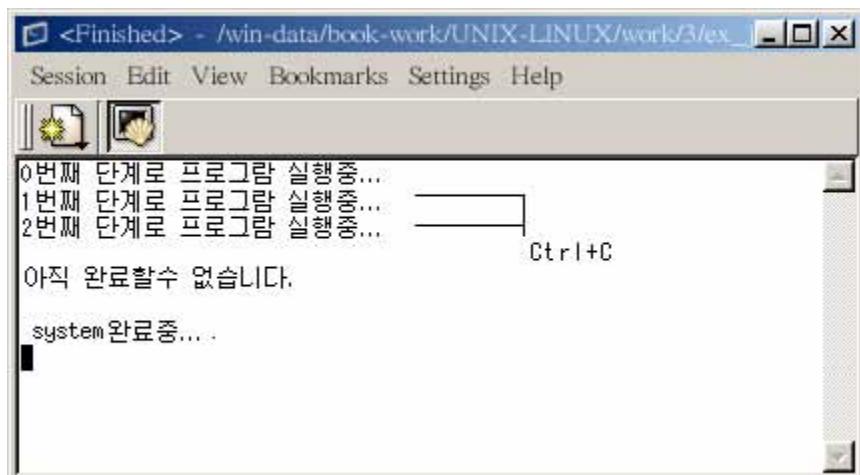



그림 3-8. ex_jmp.c의 실행결과

상식

Ipv6(Internet protocol version 6)은 어떤 규약인가

IPv6 은 현재의 표준적인 인터네트통신규약인 IPv4 를 대신하는 다음 세대의 통신규약이다. 현재의 인터네트의 주소 공간이 안고있는 여러가지 문제(클라스 B 의 고갈, 경로조종정보의 포화, 32bit 주소의 고갈 등)들을 해결하기 위하여 제정되였다. 처음에는 IPng(다음 세대의 IP)로서 연구사업이 시작되고 최종적으로 1994 년 7 월 IETF(인터네트기술개발집단)의 회의에서 SIPP16 을 기초로 한다는데 합의하여 같은 해 12 월에 표준규격을 완성하였다. IPv4 의 불필요한 기능을 삭제하고 새로운 기능을 추가한 형태로 설계되였다.

IPv4 와 비교되는 중요한 변경내용은

① 32bit 로부터 128bit 에로의 주소공간의 비약적인 확대
② 간소화된 머리부분형식
③ 경로처리 등의 고속화
④ 기능의 확장성과 유연성
⑤ 안전보호기능의 도입 등이다.

1996 년이후 실증실험을 위한 6bone 이라고 부르는 세계적인 규모의 망이 구축되고 실행을 위한 모든 검사가 진행되였다.

제4장

스레드

서론

이 장에서는 스레드를 작성하고 실행시키는데 리용되는 체계호출함수에 대해 배우게 된다. 스레드는 프로쎄스안에서의 실행단위를 말한다. Linux는 체계안에서 여러개의 프로쎄스가 동시에 실행하는 다중프로쎄스를 지원하듯이 프로쎄스안에서도 여러개의 스레드가 동시에 실행되는 다중스레드를 지원하고있다.

Linux체계는 다중프로쎄스와 다중스레드를 통하여 체계자원의 활용을 높일수 있도록 만들어준다. 동일한 프로쎄스안에 있는 스레드들은 프로쎄스가 가지고있는 자원들을 공유할수 있다. 따라서 프로쎄스들사이에서는 쉽게 주고받을수 없었던 통보문교환이나 자료공유를 쉽게 진행할수 있다.

스레드를 적절히 활용하면 하나의 프로쎄스가 여러 프로쎄스의 조합처럼 움직이도록 만들수 있다. 4장에 나오는 절들은 다음과 같다.

목표

1. 스레드에 대한 간단한 소개
2. 스레드를 위한 체계호출함수
3. 스레드프로그람작성

제1절. 스레드에 대한 간단한 소개

4.1.1. 스레드의 활용

스레드를 생성하고 기동할수 있도록 만들어준것은 개발자에게 있어서 매우 유익하다. 그것은 이것을 리용하면 여러개의 스레드가 동시에 실행되도록 만들수 있으며 이것은 류사한 작업을 동시에 수행하거나 서로 별개인 작업을 동시에 수행할수 있도록 해주기때문이다.

그러면 여러개의 프로쎄스를 실행할수 있는 다중프로쎄스가 이미 지원되는데 다중스레드처리가 어떻게 좋은가 하는 의문이 생긴다. 스레드도 프로쎄스처럼 자체의 탄창과 변수를 가질뿐아니라 프로쎄스안에 있는 대역변수나 신호조종기 또는 파일정보 등도 공유하게 된다.

따라서 프로쎄스안에서 프로쎄스를 생성하는것과 스레드들을 만드는것은 그의 사용에서 많은 차이가 있다. 단편적으로 프로쎄스들사이에는 련결된 통로를 개발자가 특별히 만들고 이것을 관리해야 하지만 스레드들은 그렇게 하지 않아도 된다. 다중프로쎄스는 개별적인 프로그람이 독립적으로 실행되는것이지만 다중스레드는 프로그람안의 함수들이 동시에 실행하는것으로 리해하면 좀 더 리해가 쉽다.

스레드의 사용과 프로쎄스의 사용은 체계에 걸리는 부하측면에서도 차이가 있다. 즉 스레드를 리용하는것이 체계의 부하를 적게 걸리게 하는 측면에서 더 우점이 있다. 이것은 프로쎄스를 생성시키는데 사용되는 부담이 스레드를 생성시키는데 사용되는 부담보다 더 크기때문이다.

체계개발측면에서도 프로쎄스(프로그람)를 하나 추가하는것과 함수를 하나 추가하는것은 차이가 있다. 스레드를 리용하여 특정한 작업을 수행하는 함수를 별도로 기동시키면 이러한 역할을 수행하는 프로쎄스와 류사한 효과를 거둘수 있다. 이때 스레드들사이에 작업결과를 공유하거나 필요한 상태검사를 하는것이 프로쎄스들사이에서 보다 훨씬 효과적이고 개발기간을 단축할수 있다.

하지만 다중프로쎄스보다 다중스레드가 항상 더 좋은것은 아니다. 특정한 스레드를 잘못 리용하면 다른 스레드에도 치명적인 영향을 미쳐 전체적으로 프로쎄스가 오동작을 할수도 있다. 이것은 여러개의 프로쎄스를 리용하는 경우보다 더 많은 문제를 발생시킨다. 그리고 다중스레드를 사용하지 않은 프로쎄스의 오유제거보다 다중스레드를 사용한 프로쎄스의 오유제거가 훨씬 더 어렵다.

다중스레드의 내부에 숨어있는 구조적인 오유는 쉽게 찾기가 힘들다. 특정한 스레드가 한번 오동작하기 시작하면 프로쎄스내부의 다른 변수들에도 영향을 미치게 되는데 이때 어떤 모듈에 의해 잘못된 값이 나오는가를 해명하기는 쉽지 않다.

그리고 개발팀의 작업분배에 있어서도 여러개의 프로쎄스를 작성하는것보다 작업분배가 복잡하다. 개발이 끝난후 작업결과에 대한 개발팀사이의 시험도 쉽지는 않다. 또한 체계의 관리에 있어서도 오동작을 일으키는 프로쎄스를 찾아서 재기동시키거나 완료시키는

것은 쉽지만 오동작을 일으키는 스레드를 찾아내서 관리하는것은 매우 어렵다.

이러한 약점을 가지고있지만 여전히 스레드는 개발과정에 널리 리용되고있으며 특히 여러가지 작업을 수행하도록 프로그람을 만들어야 하는 경우에 개발자들속에서 널리 리용되고있다.

상식

다중처리기

CPU(중앙처리장치)를 여러개 탑재하고있는것을 말한다. 매개의 CPU에 처리를 분산시킴으로써 처리성능을 높일수 있다. 주로 봉사기나 전문가작업기에서 리용된다. 그러나 다중처리기에 대응하는 CPU와 조작체계가 필요하다.

대응하는 조작체계로서는 Windows NT/XP나 OS/2, 일부 UNIX계렬의 조작체계 등이 있다. Windows 95/98은 대응하지 않는다. 또한 Microsoft BackOffice나 Oracle, Sybase, Informix 등을 리용하는 자료기지프로그람이 다중처리기에 대응하는 경우에는 그 프로그람의 준위에서도 고속화를 실현할수 있다.

4.1.2. 스레드의 고려사항

스레드가 제공되기 이전에는 다중과제처리기법을 리용하여 체계를 구축하였다. 동시에 작업해야 할 일이 없는 체계라면 특별히 다중과제처리를 고려할 필요는 없을것이다. 그러나 다양한 작업을 수행해야 하는 체계라면 동시에 여러개의 작업을 수행해야 하는 경우가 많다.

이러한 경우에 다중프로쎄스를 작성해서 체계를 기동시켜야만 했는데 프로쎄스사이에 작업공유의 어려움과 프로쎄스들의 자원공유, 그리고 프로쎄스들사이의 작업교대시간랑비 등 문제들을 해결하기 위한 방법으로서 다중스레드기법이 나오게 되였다.

다중스레드기법이 나오기 이전에는 프로쎄스안에서 하나의 스레드만이 기동하였다. 즉 main()함수를 대변하는 하나의 스레드가 main()함수의 시작과 함께 기동을 시작하고 완료와 함께 기동을 멈추는 방법이였다. 따라서 이전에는 스레드와 프로쎄스를 따로 구분할 필요가 없었다.

하지만 하나의 프로쎄스안에서 여러개의 스레드가 병렬작업을 수행하는 다중스레드에서는 프로쎄스를 스레드의 조합으로 해석할수 있다.

스레드들은 프로쎄스내부의 자원을 공유한다고 했는데 여기에는 내부의 대역변수도 해당된다. 만일 서로 다른 스레드들이 동일한 대역변수를 변경시킨다면 변수사용에서 문제가 발생하게 된다.

실례를 들어 다음의 경우를 보자.

평양역과 평성역에서 청진으로 가는 기차표를 동시에 판매하는 경우 평양역에 있는 차

표판매체계와 평성역에 있는 차표판매체계가 하나의 자료기지를 가지고 작업을 한다고 가정하자. 만일 청진행 렬차에 좌석이 하나만 남은 상태에서 평양역과 평성역에서 두명의 손님이 거의 동시에 기차표를 신청했다고 하자.

그러면 먼저 신청을 받은 평양역의 차표판매체계는 좌석수를 검사하고 남아있는 하나의 좌석에 대해 차표를 팔아 줄것이다. 이때 기차표의 처리가 끝나기도전에 이번에는 평성역의 차표판매체계가 남은 좌석을 검사한다. 아직 평양역에서 "차표판매"라는 자료처리가 끝나지 않았기때문에 좌석이 하나 남은것으로 나오게 되므로 평성역에서도 같은 자리의 차표를 판매한다.

평양역에서 처리가 끝나면 자료기지에 차표판매를 기억시킬것이며 평성역에서도 처리가 끝나면 평양역에서 처리한 자료기지에 평성역에서 처리한 자료를 덮어씌울것이다. 그러면 하나의 좌석에 나간 두장의 차표로 문제가 발생하게 되며 두 손님중 어느 한 손님은 기차를 못타게 된다.

이 과정을 스레드와 비교하여 보면 이러한 문제가 발생한 리유는 스레드가 하나의 자료기지를 완전히 처리하기도전에 또 다른 스레드가 자료기지를 사용하였기때문이다.

이런 현상이 없도록 하려면 어떤 스레드가 자료를 완전히 처리하기전에는 다른 스레드가 자료기지에 접근하지 못하도록 막아주어야 한다. 우의 실례에서 다른곳의 접근을 막는다면 평양역의 차표판매체계가 자료기지에 접근할 때 《해당 자료기지 처리중》이라는 통보문과 함께 자료기지를 사용하지 못하도록 하면 된다는것이다.

이제 평양역에서 자료기지를 사용하고 《차표판매》라고 자료처리를 끝낸 다음 평성역의 차표판매체계가 자료기지를 읽는다면 우와 같은 현상은 나타나지 않을것이다.

또한 이러한 경우 주의를 돌려야 할것은 평양역의 차표판매체계에서 처리도중 문제가 발생했는데 자료기지접근금지를 해제하지 않으면 평성역의 차표판매체계는 무한정 기다리기만 해야 한다는데 주의를 돌려야 한다. 이 실례를 통하여 다중스레드의 설계나 구현이 그렇게 쉽지는 않다는것을 알수 있다.

그러나 너무 힘들게 생각할 필요는 없다. 그것은 스레드에서 이러한 자원의 관리가 가능하도록 여러가지 기법을 제공하고있기때문에 설계 및 실천 단계에서 적절히 적용하면 되기때문이다.

이러한 리유 등으로 인하여 개발자의 능력에 따라 스레드는 프로쎄스의 성능을 높일수도 있고 성능을 낮출수도 있다.

체계를 개발할 때 다중스레드를 적용할 일이 없겠는지 미리 잘 연구해두는것이 좋다. 그것은 개발도중에 처음에는 고려하지 않았던 다중스레드기법을 적용하려면 체계의 구조를 변경해야 하는 경우가 생길수 있기때문이다. 그러나 다중스레드를 적용하면 생각지 못했던 전체 체계의 성능을 높일수도 있다. 따라서 설계나 분석단계에서 충분한 연구와 실험을 거친후에 다중스레드의 사용을 진행하여야 한다.

제2절. 스레드를 위한 체계호출함수

스레드를 위해 사용할수 있는 체계호출함수에는 Linux내부스레드와 표준으로 정의한 POSIX스레드가 있다. 지금은 대부분의 개발자들이 표준으로 정의된 POSIX스레드를 많이 사용하므로 여기서도 POSIX스레드함수들을 기본으로 설명하겠다. POSIX스레드함수를 사용하면 POSIX에 따르는 다른 가동환경에서도 동일한 API를 사용하면 되기때문에 코드를 수정하지 않아도 된다. 물론 가동환경이 달라지면 콤파일을 다시한다든가 하는 작업들은 필요한것이다. 그러나 만일 POSIX가 아닌 다른 함수를 리용하면 다른 가동환경에서의 코드변경작업은 필수적으로 제기된다.

4.2.1. pthread_create()

스레드와 관련된 체계호출중 첫번째로 보아야 할 내용이 pthread_create()함수이다. pthread_create()함수는 스레드로 실행할 모듈을 지정하고 해당 모듈을 실행시키는 작업을 수행한다. 이를 위해 사용하는 구조(struct)는 pthread_t인데 이 구조체를 리용하여 스레드를 표현하게 된다.

pthread_create는 다음과 같이 네개의 인수를 받아들이게 되는데 사용하지 않는 인수는 NULL로 처리하면 된다.

```
pthread_create(pthread_t*, pthread_attr_t*, (void*)func, (void*)arg);
```

인수로 사용된 pthread_attr_t형의 구조체는 스레드의 속성을 설정하는 값을 의미한다. 그리고 void의 지적자형변수 func는 스레드로 실행될 함수의 이름을 의미한다. 마지막으로 사용한 void의 지적자형변수 arg는 스레드에 전달할 자료가 있으면 사용한다.

pthread_create() 함수의 간단한 사용형식을 보면 다음과 같다.

```
#include <pthread.h>
void* subThread(void* arg); /*스레드로 실행될 모듈의 선언*/
pthread_t subTh_t; /* 스레드구조선언*/
pthread_create(&subTh_t, NULL, subThread, NULL);
                                        /*스레드생성 및 실행*/
```

그러면 pthread_create()를 리용하여 스레드프로그람을 작성해보자. 아래의 실례프로그람은 부분스레드를 만들어서 부분스레드와 주스레드가 각기 개별적으로 움직이면서 화면에 통보문을 출력하는 프로그람이다.

실례프로그람: pth_create.c	
1	#include <stdio.h>
2	#include <pthread.h>
3	

4	/* 새롭게 생성되여 실행될 부분스레드모듈 */
5	void* subThread(void* arg)
6	{
7	/* 부분스레드의 작업: 2s마다 통보문출력 */
8	while(1)
9	{
10	sleep(2);
11	printf("부분스레드 동작중!\n");
12	}
13	}
14	
15	/* main함수 */
16	int main()
17	{
18	/* 스레드구조선언 */
19	pthread_t subTh_t;
20	
21	/* 스레드구조와 스레드모듈을 리용하여 스레드를 생성 */
22	if(pthread_create(&subTh_t, NULL, subThread, NULL))
23	{
24	printf("부분스레드의 생성에 실패\n");
25	return 0;
26	}
27	
28	/* 주(main)스레드의 작업: 1s마다 통보문출력 */
29	while(1)
30	{
31	sleep (1);
32	printf("주스레드 동작중!\n");
33	}
34	return 1;
35	}
36	

우의 실례프로그람을 보면 주스레드와 부분스레드가 모두 무한순환을 하고있다는것을 알수 있다. 프로그람을 다중스레드로 만들지 않으면 동시에 두개의 모듈을 무한순환시

키는것은 힘들지만 다중스레드를 만들면 어렵지 않게 할수 있다.

우의 프로그람을 콤파일하여 실행해보자. 그림 4-1에서는 주스레드와 부분스레드가 모두 무한순환하고있으므로 특정한 스레드가 먼저 완료되는 경우를 확인할수 없다.

그렇다면 우의 실례에서 만일 주스레드가 완료된다면 어떻게 되겠는가? 실례프로그람에는 특별히 스레드를 완료시키는 부분이 없기때문에 주스레드가 완료되면 전체 프로쎄스도 완료된다.

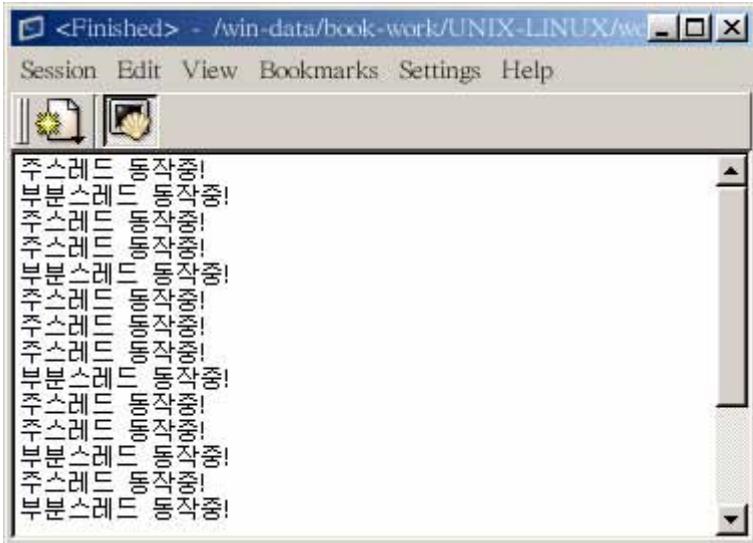



그림 4-1. pth_create.c 의 실행결과

4.2.2. pthread_exit()와 pthread_self()

프로쎄스가 완료되면 내부에 있는 스레드들은 모두 완료되게 된다. 이때 주스레드만 완료가 되고 프로쎄스에는 문제가 없도록 하려면 어떻게 해야 하겠는가?

만일 그렇게만 된다면 주스레드의 완료와 무관계하게 부분스레드들이 각자에게 맡겨진 작업을 수행할수 있을것이다.

> **알아둡시다**
>
> 부분스레드의 경우에는 특별한 조작이 없이 실행이 중지되여도 프로쎄스는 완료되지 않는다.

이런 경우 pthread_exit()를 리용하여 주스레드만 완료가 되고 프로쎄스는 완료되지 않도록 할수 있다. pthread_exit()함수는 프로쎄스의 완료를 위해 사용하는 exit()함수

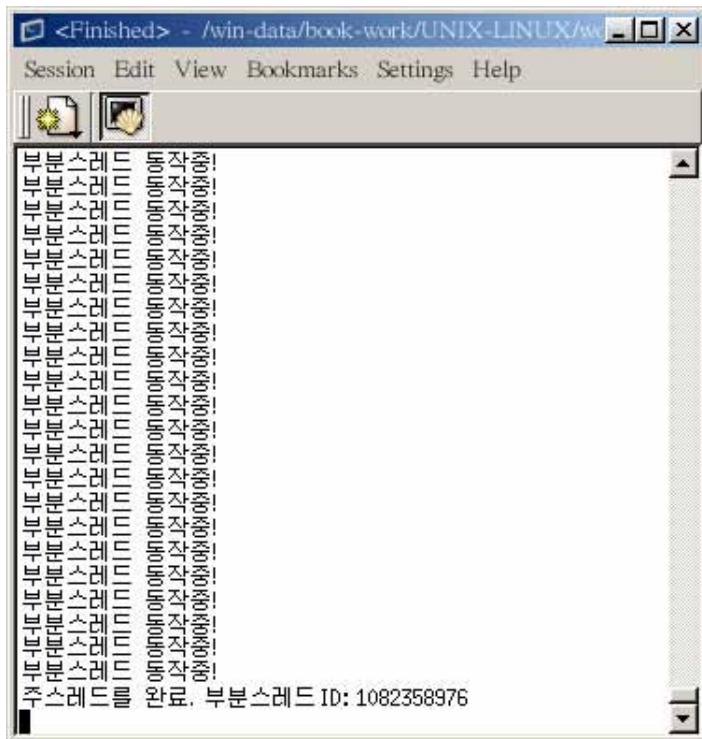
와 비슷한데 다음과 같이 간단하게 사용할수 있다.

```
pthread_exit (0) ;
```

pthread_exit에서 리용되는 인수의 설명과 사용방법은 pthread_join()에 대하여 해설하면서 자세히 설명하기로 하자.

프로세스가 ID를 가지고있는것처럼 스레드도 ID를 가지고있으며 이것을 알아볼수 있다. 이때 사용되는 체계호출함수는 pthread_self()이다. pthread_self()함수가 반환하는 값을 리용하여 스레드의 ID를 얻을수 있다.

그러면 pthread_exit()와 pthread_self()를 리용한 실례프로그람을 작성해보자. 아래의 실례프로그람에서는 주스레드는 완료되고 부분스레드만 실행되도록 만든 프로그람이다. 그리고 부분스레드가 실행되면 부분스레드의 ID를 화면에 현시한다.

실례프로그람: pth_getid.c	
1	#include <stdio.h>
2	#include <pthread.h>
3	
4	/* 새롭게 생성되여 실행될 부분스레드 모듈 */
5	void* subThread(void* arg)
6	{
7	/* 부분스레드의 번호를 출력 */
8	int subThID = pthread_self();
9	printf("부분스레드 번호: %d\n",subThID);
10	
11	/* 부분스레드 작업: 매초마다 통보문 출력 */
12	while(1)
13	{
14	sleep(1);
15	printf("부분스레드 동작중!\n");
16	}
17	}
18	
19	/* 프로세스의 main함수 */
20	int main()
21	{
22	/* 스레드 구조 선언 */
23	pthread_t subTh_t;
24	/* 주스레드의 번호를 출력 */

다중스레드로 작업을 하게 되면 특정한 스레드의 완료를 기다려야 하는 경우가 있다. 그리고 해당 스레드의 작업결과를 보고 작업을 다시 하던지 아니면 다음 작업으로 넘어가야 하는지를 결정해야 할 경우도 있다.

4.2.3. pthread_join()

pthread_join()함수를 리용하면 특정한 스레드가 완료될 때까지 실행을 멈추도록 할 수 있다. 그리고 해당 스레드가 완료되면서 되돌려준 값을 받아서 검사를 진행한다.

먼저 pthread_join의 사용형식은 다음과 같다.

```
int pthread_join (pthread_t, void**);
```

여기서 pthread_t는 완료를 기다리는 스레드를 지정하고 void**는 해당 스레드가 완료되면서 사용한 값을 얻어오는데 사용된다. 이때 해당 스레드는 값을 넘겨주기 위해 pthread_exit()를 사용하게 된다.

례를 들어 스레드가 "pthread_exit((void *)1)"를 리용하여 완료하게 되면 pthread_join에는 1에 대한 지적자가 전달된다. 따라서 스레드가 완료될 때 성공했을 경우와 실패했을 경우에 따라 exit값을 다르게 주면 join을 리용하여 스레드의 작업수행결과를 확인할수 있다.

직접 실례프로그람을 통해 이를 확인해보자.

아래의 실례프로그람에서는 스레드를 만든다음 pthread_join을 리용하여 스레드가 완료될 때까지 대기하며 그 다음 스레드가 완료되면서 보내온 값을 받아서 화면에 출력하게 된다.

실례프로그람: pth_join.c	
1	#include <pthread.h>
2	
3	/* 새롭게 생성되여 실행될 부분스레드 모듈 */
4	void* subThread(void* arg)
5	{
6	printf("부분스레드의 동작을 시작!\n");
7	sleep(1);
8	printf("부분스레드의 동작을 완료!\n");
9	pthread_exit((void *)1);
10	}
11	
12	/* main함수 */

```
13  int main()
14  {
15          /* 스레드구조선언 */
16          pthread_t subTh_t;
17          int *pstsVal;
18
19          /* 스레드구조와 스레드모듈을 리용하여 스레드를 생성 */
20          if(pthread_create(&subTh_t, NULL, subThread, NULL))
21          {
22                  printf("부분스레드의 생성에 실패\n");
23                  return 0;
24          }
25
26          /* pthread_join을 리용하여 부분스레드의 완료를 검사 */
27          pthread_join(subTh_t, (void **)&pstsVal);
28
29          /* pstsVal값을 통해 완료상태를 출력 */
30          printf("pstsval: %d\n",pstsVal);
31          return 1;
32  }
```

그림 4-3에서 프로그람의 실행결과를 보면 pstsval변수에 스레드에서 넘겨준 값이 그대로 할당되였다는것을 알수 있다.

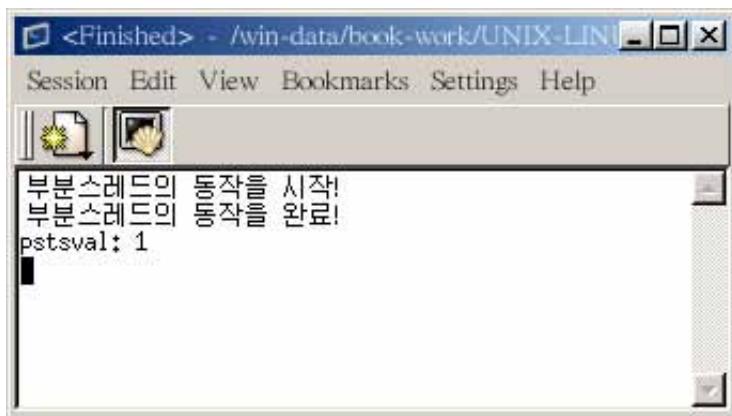



그림 4-3. pth_join.c 의 실행결과

우의 실레를 통하여 부분스레드가 주스레드에 간단한 정보를 전달한것을 볼수 있다. 이번에는 스레드를 생성하여 실행시킬 때 특정한 자료 등을 인수로 전달하는것을 보자.

4.2.4. 통보문전달

스레드를 만들어 실행할 때 특정한 자료를 전달해야 할 경우가 많다. 스레드는 프로쎄스안의 자원을 공유하기때문에 대역변수를 리용하면 어렵지 않게 스레드사이에 자료를 쉽게 주고받을수 있을것이다. 하지만 대역변수의 사용은 고려하는것이 좋으며 모든 자료를 대역변수로 만들수도 없다.

이때에는 스레드에 원하는 자료를 인수로 넘겨주는것이 좋다. 앞에서 pthread_create() 함수의 사용형식을 소개할 때 다음과 같이 void*형의 arg를 보았다.

```
pthread_create ( pthread_t*, pthread_attr_t*, (void*)func, (void*)arg );
```

이렇게 void의 지적자형변수를 리용하여 원하는 형의 자료를 주고받으면 되는데 이때 사용하는 형이 void의 지적자형변수이기때문에 특별히 정해진 형이 없다. 따라서 사용자가 원하는 구조체를 만들어 전달할수도 있다.

례를 들어 소케트를 리용하여 외부체계와 콤퓨터망접속을 하려고 하는데 이것을 스레드로 처리한다고 하자. 이를 위해 스레드에 접속하려는 체계정보를 전달한다. 그러면 해당 정보를 받은 스레드는 이것을 리용하여 콤퓨터망의 접속과 자료전송을 담당하게 된다. 그러면 이러한것을 바탕으로 하는 실레프로그람을 만들어보자. 여기에서 스레드에 전달하려고 하는 자료의 형은 다음과 같다.

```
typedef struct
{
char *ipAddr; /* IP 주소 */
char *hostName; /* 장비이름 */
int portNo; /* 포구번호 */
} IpInfoType;
```

이것을 void*형으로 강제형변환하여 스레드에 전달하게 된다. 그러면 스레드는 void*형으로 들어온 인수를 IpInfoType*형으로 또 다시 강제형변환을 한 다음에야 해당 정보를 꺼내올수 있다. 이것을 구현한 실레프로그람을 작성하자.

실레프로그람: pth_msg.c	
1	#include <stdio.h>
2	#include <pthread.h>
3	
4	/* 스레드에게 전달할 구조체 */
5	Typedef struct

6	{
7	char *ipAddr; /* IP주소 */
8	char *hostName; /* 장치이름 */
9	int portNo; /* 포구번호 */
10	} IpInfoType;
11	
12	/* 부분스레드모듈 */
13	void *setConnect (void *ipInfo)
14	{
15	/* 강제형변환 작업: void -> IpInfoType */
16	IpInfoType *connInfo = (IpInfoType*)ipInfo;
17	
18	/* 구조체의 각 정보를 화면에 출력 */
19	printf("setconnect 스레드 실행\n");
20	printf("ipaddress: %s\n",connInfo->ipAddr);
21	printf("hostname: %s\n",connInfo->hostName);
22	printf("portno: %d\n",connInfo->portNo);
23	printf("해당 system으로 접속중…\n");
24	sleep(1);
25	}
26	/* 프로쎄스의 main함수 */
27	int main()
28	{
29	/* 스레드 선언 */
30	pthread_t setConnect_t;
31	
32	/* IpInfoType구조체를 생성한 다음 값을 입력 */
33	IpInfoType ipInfo;
34	ipInfo.ipAddr = "192.168.8.100";
35	ipInfo.hostName = "kkk";
36	ipInfo.portNo = 9000;
37	
38	/* 부분스레드를 생성. 이때 ipInfo를 void형으로 형변환 후 전달 */

39	if(pthread_create(&setConnect_t, NULL, setConnect, (void *)&ipInfo))
40	{
41	printf("setConnect_t스레드의 생성에 실패!\n");
42	return 0;
43	}
44	pthread_exit(0);
45	}

우와 같이 프로그람작성이 끝났으면 콤파일을 진행하고 실행을 시켜보자. 프로그람의 실행을 통해 주스레드에서 전달한 내용을 부분스레드가 받아 활용하는것을 볼수 있다.

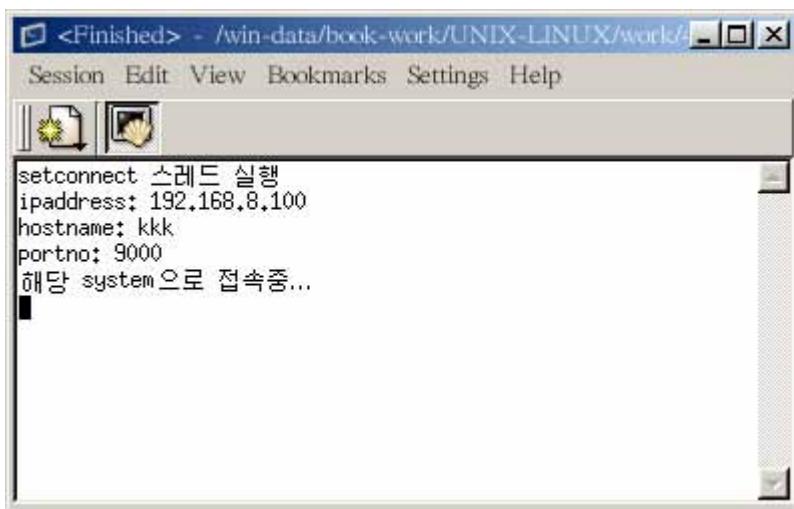



그림 4-4. pth_msg.c의 실행결과

알아둡시다

스레드에 자료를 전달하기 위한 알맞는 구조체를 적당히 정의한 다음에는 프로쎄스에서 생성한 모든 스레드에 일률적으로 전달하는것이 좋다. 실례로 체계정보나 다른 스레드에 대한 상태를 전달해서 스레드의 초기화나 마지막에 활용할수 있도록 만들수 있다.

4.2.5. pthread_attr

pthread_create()함수의 사용형식에 대한 설명에서 스레드의 속성을 변경시키기 위하여 사용하는 구조체로 pthread_attr_t가 있다고 했다. pthread_attr_t는 해당한 특성을 설정하고 스레드를 생성할 때 함께 사용한다. 이때 사용되는 체계호출함수에는 pthread_attr_xxx()가 있는데 pthread_attr_xxx()함수의 간단한 사용형식을 보면 다음과 같다.

```
pthread_attr_t subAttr_t; /* attr_t형의 subAttr_t선언 */
pthread_attr_init(&subAttr_t); /* subAttr_t 변수초기화 */
pthread_attr_setXXX(&subAttr_t, … ); /*subAttr_t변수에 값을 설정 */
pthread_attr_getXXX(&subAttr_t, … ) ; /* subAttr_t 변수의 값을 조회*/
pthread_create( … , &subAttr_t, … , NULL);/* subAttr_t 를 리용하여
스레드를 생성 */
pthread_attr_destroy(&subAttr_t); /* destroy()를 통해 subAttr_t 제거 */
```

그러면 스레드의 속성을 설정하고 이것을 리용하여 스레드를 생성하는 실례프로그람을 만들어보자. 아래의 실례프로그람에서는 탄창크기를 조절하는 pthread_attr_t구조체를 작성한 다음 이것을 스레드생성에 적용한 실례를 보여주고있다. 이를 위해 사용된 함수는 다음과 같다.

```
pthread_attr_setstacksize(); pthread_attr_getstacksize ( );
```

그러면 원천코드를 보도록 하자.

실례프로그람: pthread_attr

1	#include <pthread.h>
2	
3	/* attr_t형의 대역변수 subAttr_t 선언 */
4	pthread_attr_t subAttr_t;
5	
6	/* 새로 생성되여 실행될 부분스레드 모듈 */
7	void* subThread(void *arg)
8	{
9	/* 기억기크기의 변수선언 */
10	size_t memSize;
11	/* 탄창크기를 얻은 다음 출력 */
12	pthread_attr_getstacksize(&subAttr_t, &memSize);
13	printf("부분스레드 attr의 탄창크기: %d\n",memSize);
14	pthread_exit(0);
15	}
16	
17	/* main함수 */
18	int main()

```
19 {
20         /* 스레드구조와 기억기크기 변수선언 */
21         pthread_t subTh_t;
22         size_t memSize;
23 
24         /* init() 함수를 리용한 subAttr_t의 초기화 */
25         pthread_attr_init(&subAttr_t);
26 
27         /* 초기화된 subAttr_t의 탄창기억기크기를 가져오기 */
28         pthread_attr_getstacksize(&subAttr_t, &memSize);
29         printf("attr의 초기탄창크기: %d\n",memSize);
30 
31         /* subAttr_t의 탄창기억기크기 설정후 다시 가져오기 */
32         pthread_attr_setstacksize(&subAttr_t, 1024*3);
33         pthread_attr_getstacksize(&subAttr_t, &memSize);
34         printf("주스레드 attr의 탄창크기: %d\n",memSize);
35 
36         /* 스레드구조와 스레드모듈을 리용하여 스레드 생성 */
37         if(pthread_create(&subTh_t, &subAttr_t, subThread, NULL))
38         {
39                 printf("부분스레드의 생성에 실패\n");
40                 return 0;
41         }
42 
43         /* pthread_join을 리용하여 부분스레드의 완료때까지 대기 */
44         pthread_join(subTh_t, NULL);
45 
46         /* destroy()를 통해 subAttr_t 제거 */
47         pthread_attr_destroy(&subAttr_t);
48         return 1;
49 }
```

프로그람이 실행되면 대역변수로 선언된 pthread_attr_t구조체가 가지고있는 탄창크기를 주스레드와 부분스레드에서 각각 현시한 결과를 볼수 있다.(그림 4-5)

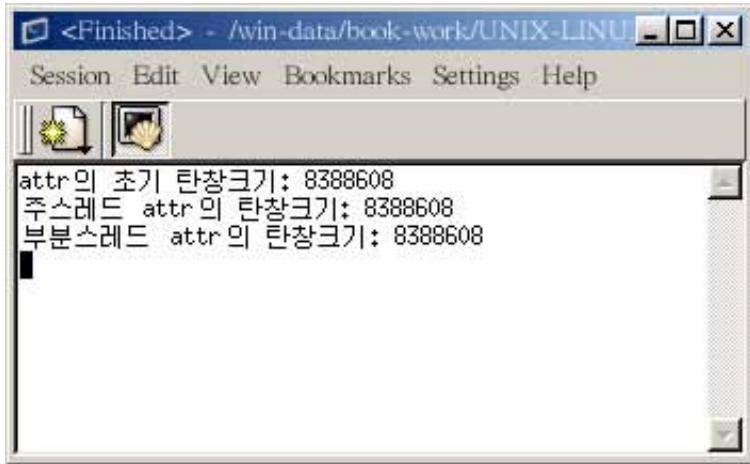



그림 4-5. pth_attr.c 의 실행결과

상식

다중표시장치(multi-display)

1대의 개인용콤퓨터에 2개이상의 영상카드와 표시장치를 접속하여 동시에 현시하는것을 말한다. 탁상출판과 같이 한대의 표시장치로는 다 볼수 없는 정도의 큰 편집화면이 요구되는 분야에서 필요한 기능이지만 오락 등에서 활용되는 경우도 많다. 최근에는 콤퓨터의 능력이 비약적으로 높아져 이전에는 여러대의 콤퓨터로 해야 하던 업무관리나 사무행정관리를 한대의 콤퓨터로도 할수 있는 기술조건이 갖추어지는데 맞게 여러 부분들에서 다중표시장치를 널리 받아들이는 추세를 보이고있다.

제3절. 스레드 프로그람작성

이 절에서는 앞에서 소개하지 않았던 함수들을 리용하여 스레드에 좌물쇠를 잠그거나 해제하는 방법을 숙련해보며 또한 C언어가 아닌 C++언어를 사용하여 스레드를 구현한 실례를 통하여 스레드에 대해 더 깊이 보도록 한다.

4.3.1. mutex

다중스레드로 동작하는 체계에서는 다중프로쎄스에서와 같이 동기화문제가 발생하게 된다. 이것은 여러개의 스레드가 동시에 특정한 령역을 사용할 때 문제로 제기되는데 이것을 제대로 해결하지 못하면 잘못된 자료를 사용하는 결과를 가져오게 된다.

mutex를 소개하기전에 동일한 령역을 사용하면 어떤 문제가 발생할수 있겠는가에 대해 간단한 실례를 통하여 보도록 하자. 실지 프로그람의 작성과 리용에서는 이 보다 더 치

명적인 문제가 수시로 발생할수 있기때문에 상당한 주의를 돌릴 필요가 있다.

아래의 실례프로그람은 대역변수를 리용하여 접속할 체계의 포구번호를 설정하는 프로그람이다. 체계정보를 설정하는 함수가 존재하는데 주스레드와 부분스레드가 각각 이 함수를 호출하여 포구번호를 얻어오게 된다. 포구번호를 얻고나면 대역변수는 하나씩 증가되고 이것을 리용하여 스레드들은 또 다른 포구번호를 활용하게 된다. 여기서 주의할것은 스레드들이 동일한 포구번호를 사용하지 않도록 만드는것이다. 동일한 포구를 사용하면 해당 스레드에 문제가 제기될뿐만아니라 전체 체계의 동작에도 문제가 발생할수 있다. 그러면 먼저 원천파일을 보도록 하자. 원천코드를 보면서 어떤 곳에서 어떤 문제가 생기겠는가에 대해서 생각해보자.

실례프로그람: pth_nomutex.c	
1	#include <stdio.h>
2	#include <pthread.h>
3	
4	/* 스레드가 사용할 IpInfoType 선언 */
5	typedef struct
6	{
7	char *ipAddr; /* IP주소 */
8	char *hostName; /* 장치이름 */
9	int portNo; /* 포구번호 */
10	} IpInfoType;
11	
12	/* 포구번호에 사용될 계수기 선언 */
13	int countNo = 0;
14	
15	/* IpInfoType에 값을 넣은후 지적자(pointer)를 반환하는 함수 */
16	IpInfoType *get_ipInfo(void)
17	{
18	/* IpInfoType구조체생성후 값을 입력 */
19	IpInfoType ipInfo;
20	int mutexRlt;
21	
22	/* 계수기증가후 3s 휴식 */
23	countNo++;

24	sleep(3);
25	
26	/* ipInfo구조체에 값을 입력 */
27	ipInfo.ipAddr = "192.168.8.100";
28	ipInfo.hostName = "kkk";
29	ipInfo.portNo = countNo;
30	
31	return &ipInfo;
32	}
33	
34	/* 부분스레드모듈. 접속대상장치의 포구를 입수 */
35	void *setConnect(void *arg)
36	{
37	IpInfoType *subInfo;
38	printf("SETCONNECT스레드실행\n");
39	
40	/* get_ipInfo()함수를 매초마다 호출. 포구번호 출력 */
41	while(1)
42	{
43	subInfo = get_ipInfo();
44	/* 포구번호를 화면에 출력 */
45	printf("부분스레드가 가진 포구번호: %d\n",subInfo->portNo);
46	sleep(1);
47	}
48	}
49	
50	/* 프로쎄스의 main함수 */
51	int main()
52	{
53	/* 스레드선언 */
54	pthread_t setConnect_t;
55	IpInfoType *mainInfo;
56	

```
57        /* 부분스레드 생성 */
58        if(pthread_create(&setConnect_t, NULL, setConnect, NULL))
59        {
60                printf("setConnect_t스레드의 생성에 실패!\n");
61                return 0;
62        }
63
64        /* get_ipInfo()함수를 매초마다 호출. 포구번호 출력 */
65        while(1)
66        {
67                mainInfo = get_ipInfo();
68                printf("주스레드가 가진 포구번호: %d\n",mainInfo->portNo);
69                sleep(1);
70        }
71 }
```

원천파일을 분석하는 과정에 특별한 문제점을 찾지 못하였다면 원천코드를 콤파일하고 실행시켜보자. 실행결과는 그림 4-6과 같다.

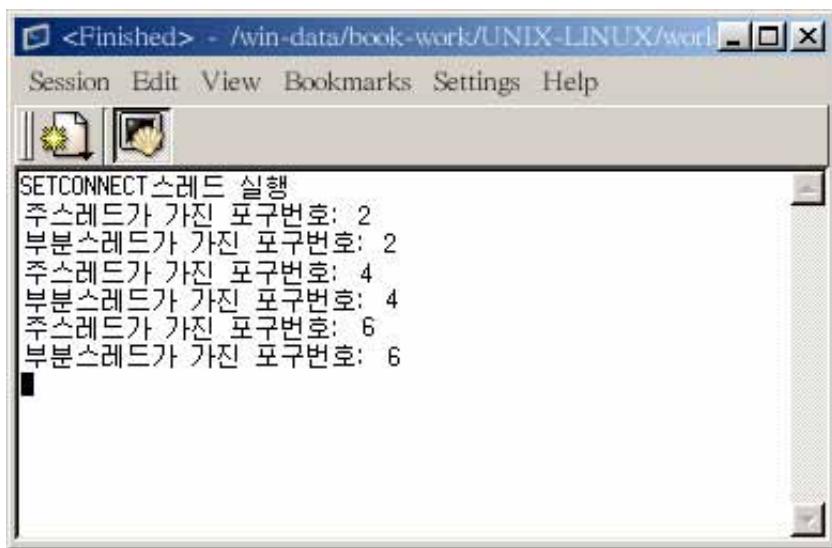



그림 4-6. pth_nomutex.c의 실행결과

프로그람의 실행결과에서 주스레드와 부분스레드는 동일한 포구번호를 계속 할당받았다는것을 알수 있다. 이것은 두개의 스레드가 거의 동시에 대역변수를 증가시키고 할당을 받았기때문이다. 이러한 문제는 쉽게 발생하며 쉽게 찾기 어려운 오유이다.하지만 스

레드에서는 이러한 문제를 근본적으로 해결할수 있는 기법들을 제공하고있다.

스레드의 동기화문제를 해결하기 위해 가장 많이 활용되는 기법이 바로 mutex이다. mutex는 뒤에서 소개될 신호기와 류사하며 특정한 령역을 하나의 스레드만이 사용할수 있도록 잠금(lock)과 잠금해제(unlock)를 적용함으로써 그의 리용방법은 아주 간단하다. 즉 mutex를 lock시키면 다른 스레드는 unlock될 때까지 계속 대기해야 한다. 그러다가 unlock시키면 대기하던 스레드는 그곳에 접근할수 있다. 이때 lock시킨 스레드만이 unlock를 할수 있기때문에 특정한 령역에 대하여 오직 하나의 스레드만이 접근하는 원칙을 지킬수 있다.

그러면 먼저 mutex를 사용하기 위해 제공되는 함수와 구조(struct)를 살펴보자. 아래의것은 이러한 함수들의 간단한 사용형식들이다.

```
pthread_mutex_t mx_t ;
int pthread_mutex_init (pthread_mutex_t *, const pthread_mutexattr_t *;
int pthread_mutex_lock (pthread_mutex_t *);
int pthread_mutex_unlock(pthread_mutex_t * );
int pthread_mutex_destroy(pthread_mutex_t * );
```

먼저 mutex를 사용하기 위해서는 pthread_mutex_t형의 변수를 선언하고 pthread_mutex_init함수를 통해 초기화과정을 거쳐야 한다. 만일 함수를 실행하지 않고 초기화를 시키려면 다음과 같이 한다.

```
pthread_mutex_t mx_t = PTHREAD_MUTEX_INITIALIZER ;
```

pthread_mutex_lock()함수는 선언된 mutex를 잠그는 작업을 수행하고 반대로 pthread_mutex_unlock()함수는 잠긴 mutex를 해제하는 작업을 수행한다. 마지막으로 pthread_mutex_destroy()함수는 선언된 mutex를 제거하는 작업을 수행한다. 개개의 함수들은 성공하면 0을 되돌리며 실패하면 오유코드를 되돌리게 된다.

그러면 앞에서 설명했던 실례프로그람에 mutex를 적용해보도록 하자. mutex를 적용할곳은 포구번호로 사용할 대역변수의 값을 조작하는곳인 get_ipInfo()함수의 내부가 된다. 아래에 수정된 원천코드를 보여주고있다.

실례프로그람: pth_mutex.c	
1	#include <stdio.h>
2	#include <pthread.h>
3	
4	/* 스레드가 사용할 IpInfoType 선언 */
5	typedef struct

6	{
7	char *ipAddr; /* IP 주소 */
8	char *hostName; /* 장치이름 */
9	int portNo; /* 포구번호 */
10	} IpInfoType;
11	
12	/* mutex getInfo 선언 */
13	pthread_mutex_t getInfo = PTHREAD_MUTEX_INITIALIZER;
14	
15	/* 포구번호에 사용될 계수기선언 */
16	int countNo = 0;
17	
18	/* IpInfoType에 값을 넣은후 지적자(pointer)를 반환하는 함수 */
19	IpInfoType *get_ipInfo(void)
20	{
21	/* IpInfoType구조체생성후 값을 입력 */
22	IpInfoType ipInfo;
23	int mutexRlt;
24	
25	/* mutex lock 시작 */
26	mutexRlt = pthread_mutex_lock(&getInfo);
27	/* 계수기 증가후 3s 휴식 */
28	countNo++;
29	sleep(3);
30	
31	/* ipInfo구조체에 값 입력 */
32	ipInfo.ipAddr = "192.168.8.100";
33	ipInfo.hostName = "kkk";
34	ipInfo.portNo = countNo;
35	
36	/* mutex lock 해제 */
37	mutexRlt = pthread_mutex_unlock(&getInfo);
38	return &ipInfo;

```
39  }
40  /* 부분스레드모듈. 접속대상장치의 포구를 입수 */
41  void *setConnect(void *arg)
42  {
43          IpInfoType *subInfo;
44          printf("SETCONNECT스레드 실행\n");
45  
46          /* get_ipInfo()함수를 매초마다 호출. 포구번호출력 */
47          while(1)
48          {
49                  subInfo = get_ipInfo();
50                  /* 포구번호를 화면에 출력 */
51                  printf("부분스레드가 가진 포구번호: %d\n",subInfo->portNo);
52                  sleep(1);
53          }
54  }
55  
56  /* 프로쎄스의 main 함수 */
57  int main()
58  {
59          /* 스레드 선언 */
60          pthread_t setConnect_t;
61          IpInfoType *mainInfo;
62  
63          /* 부분스레드 생성 */
64          if(pthread_create(&setConnect_t, NULL, setConnect, NULL))
65          {
66                  printf("setConnect_t스레드의 생성에 실패!\n");
67                  return 0;
68          }
69  
70          /* get_ipInfo()함수를 매초마다 호출. 포구번호출력 */
71          while(1)
```

72	{
73	mainInfo = get_ipInfo();
74	printf("주스레드가 가진 포구번호: %d\n",mainInfo->portNo);
75	sleep(1);
76	}
77	}

원천코드의 수정을 진행하였으면 콤파일을 한다음 프로그람을 실행시켜보자. 그림 4-7은 프로그람의 실행결과를 보여주고있다.

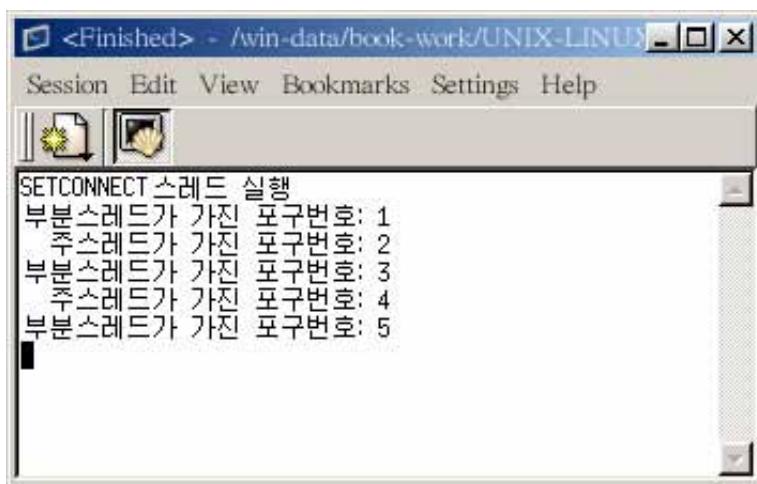



그림 4-7. pth_mutex.c의 실행결과

우의 실행결과를 통해 스레드들이 서로 다른 포구번호를 순차적으로 사용하고있음을 알수 있다. 지금까지 설명한 mutex는 적용할만한 부분들이 상당히 많다. 하지만 mutex를 잘못 설정해서 사용하면 또 다른 문제점이 생길수 있다. 실례를 들어 lock를 실행하고 unlock를 실행하지 않아 특정한 령역이 계속 잠겨있는 경우도 발생할수 있다. 이처럼 새로운 기능을 추가로 도입한후에는 새로운 문제가 발생할수 있으므로 주의를 돌려 생길수 있는 문제들에 관심을 돌려야 한다.

4.3.2. 스레드 조건변수

mutex는 특정한 령역에 대하여 lock와 unlock를 실행하면서 목적하는 작업을 수행하게 되는데 여기에 신호를 기다리는 기능과 신호를 발생시키는 기능을 추가하여 스레드의 동기화를 보다 능동적으로 수행할수 있다.

mutex의 경우에는 mutex내부에서 조건에 따라 lock와 unlock를 실행하기는 쉽지 않다. 그리고 다른 스레드가 언제 mutex를 해제했는지 그 시점을 확인하기도 쉽지 않다. 하지만 조건에 따라 mutex를 잠그고 해제할수 있으며 신호를 리용하여 mutex가 해제되였음을 다른 스레드에 알릴수 있다면 많은 문제를 해결할수 있을것이다.

이러한 기능을 리용하면 특정한 스레드의 작업의 탈퇴와 함께 다른 스레드의 작업이 보다 원활히 진행될수 있을것이다. 이것을 가능하게 만드는것이 바로 스레드조건변수로 되는 pthread_cond이다.

pthread_cond내부에서는 신호를 발생시키거나 신호를 기다리는 기능을 제공하고있다. 이때 말하는 신호는 프로쎄스가 사용하는 일반적인 의미의 신호와는 다르다. 신호를 사용하려면 먼저 pthread_cond_t형의 조건변수를 생성해야 하며 조건변수를 초기화해야 한다. 초기화하는 방법은 mutex처럼 init함수를 사용하거나 선언자를 리용하면 된다.

```
pthread_cond_t condT;
pthread_cond_t condT=PTHREAD_COND_INITIALIZER;
pthread_cond_init (&condT);
```

생성된 스레드조건변수를 리용하여 신호를 보내는 함수에는 다음과 같은것들이 있다.

```
int pthread_cond_signal(pthread_cond_t *);
int pthread_cond_broadcast(pthread_cond_t *);
```

signal함수와 broadcast함수는 모두 해당 조건변수에 신호를 보낸다. 차이점은 signal 함수는 해당되는 스레드에만 신호를 전송하고 broadcast함수는 모든 스레드에 신호를 전송하는것이다. 만일 신호를 전송했는데 수신대기중인 스레드가 없으면 신호는 무시된다.

스레드조건변수와 mutex를 리용하여 전송될 신호을 기다리는 함수에는 다음과 같은 것들이 있다.

```
pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *);
pthread_cond_timedwait(pthread_cond_t*, pthread_mutex_t*, const
struct timespec*);
```

wait()함수를 사용하면서 보면 조건변수뿐만 아니라 mutex도 함께 인수로 사용하는 것을 볼수 있다. 이러한 wait함수가 제대로 수행되려면 내부에 사용된 mutex가 초기화되여야 하며 lock되여 있어야 한다. wait()함수가 실행되면 함수는 내부에서 사용된 mutex의 lock를 해제한채로 대기작업에 들어간다. 그리고 이 함수들은 조건변수에 대한 신호가 검출되면 작업을 재개한다. 즉 해당한 조건변수에 대한 신호가 발생하면 내부의 인수로 지정된 mutex를 잠근후 wait()함수의 다음 행을 실행하게 된다. wait함수가운데서 timedwait()함수는 인수로 사용된 timespec에 지정한 시간만큼 대기를 진행하게 된다.

만일 지정된 시간안에 신호가 검출되지 않으면 시간초과오유코드와 함께 실행이 중지된다. 스레드 조건변수의 활용이 끝났으면 destroy()함수를 호출하여 조건변수를 제거할수 있다. destroy()함수의 사용형식은 다음과 같다.

```
int pthread_cond_destroy(pthread cond t *);
```

그러면 이제 이러한 신호조건변수와 mutex를 활용한 실례프로그람을 만들어보자. 작성할 실례는 두개의 부분스레드가 체계내부에 있는 계수기를 활용하는 문제이다. 이때 계수기를 증가시키는 함수가 따로 존재하며 이 함수를 스레드들이 호출하게 된다. 매개 스레드는 다른 스레드가 계수기 증가함수를 사용하는동안 대기상태에 들어간다.

그러다가 상대스레드가 신호를 전송하면 계수기증가함수를 호출한 다음 현재의 계수기수를 화면에 현시하게 된다. 이를 위해 먼저 다음과 같이 mutex와 스레드조건변수를 선언하고 초기화하도록 한다.

```
  pthread_mutex_t thread1Mx =
PTHREAD_MUTEX_INITIALIZER;
  pthread_mutex_t thread2Mx =
PTHREAD_MUTEX_INITIALIZER;
  pthread_cond_t checker = PTHREAD_COND_INITIALIZER;
```

그 다음에는 두개의 스레드를 생성한다. 먼저 생성된 스레드는 mutex를 lock시킨다음 계수기증가함수를 호출하게 된다. 그 다음 신호함수를 리용하여 조건변수에 신호를 전송한다. 다음에는 다른 스레드의 신호를 기다리기 위해 wait함수를 리용하여 대기작업에 들어간다. 매 작업의 순서는 다음과 같다.

```
pthread_mutex_lock(&thread1Mx); /* mutex lock */
setCount(); /* 계수기 증가 함수호출 */
pthread_cond_signal(&checker); /* 스레드조건변수에 신호전송 */
pthread_cond_wait(&checker, &thread1Mx); /* 신호 대기 */
```

첫번째 스레드와 쌍으로 작업할 두번째 스레드에 대해 살펴보자. 두번째 스레드는 첫번째 스레드와 순서가 다르게 작업을 수행한다. 즉 mutex를 lock시킨 다음 신호를 기다리는 대기함수를 실행시킨다. 신호를 받게 되면 계수기증가함수를 호출한 다음 스레드조건변수에 신호를 전송하게 된다. 즉 다음과 같이 실행한다.

```
pthread_mutex_lock(&thread2Mx); /* mutex lock */
pthread_cond_wait(&checker, &thread2Mx); /* 신호대기 */
setCount();  /* 계수기 증가함수 호출 */
Pthread_cond_signal(&checker); /* 스레드조건변수에 신호전송 */
```

마지막으로 스레드의 작업이 모두 완료되였으면 다음과 같이 선언된 mutex와 스레드의 조건변수를 체계에서 제거(destroy)하도록 한다.

```
pthread_mutex_destroy (&thread1Mx);
pthread_mutex_destroy (&thread2Mx);
pthread_cond_destroy (&checker);
```

그러면 지금까지 설명한 내용에 기초하여 pth_mutex와 같은 역할을 하는 프로그람을 작성하여 보자. 아래에 원천코드를 서술하였다.

실례프로그람: pth_cond.c	
1	#include <stdio.h>
2	#include <pthread.h>
3	
4	/* mutex_t와 cond_t 선언 및 초기화 */
5	pthread_mutex_t thread1Mx = PTHREAD_MUTEX_INITIALIZER;
6	pthread_mutex_t thread2Mx = PTHREAD_MUTEX_INITIALIZER;
7	pthread_cond_t checker = PTHREAD_COND_INITIALIZER;
8	
9	/* 대역변수로 사용할 계수기선언 */
10	int countNo = 0;
11	
12	/* 계수기를 증가시키는 함수 */
13	void setCount(void)
14	{
15	/* 계수기 증가 */
16	countNo++;
17	sleep(1);
18	}
19	
20	/* 첫번째 스레드모듈 */
21	void *runThread1(void *arg)
22	{
23	printf("첫번째 스레드 실행\n");
24	/* setCount()함수를 매초마다 호출 */
25	while(1)
26	{
27	/* 계수기가 5를 넘으면 스레드 완료 */
28	if(countNo >= 5) pthread_exit(0);
29	/* thread1 mutex lock */
30	pthread_mutex_lock(&thread1Mx);

31	/* setCount()실행후 count번호를 출력 */
32	setCount();
33	printf("첫번째 스레드가 얻어온 계수기 번호: %d\n",countNo);
34	/* checker신호발생후, wait수행 */
35	pthread_cond_signal(&checker);
36	pthread_cond_wait(&checker, &thread1Mx);
37	/* thread1 mutex lock 해제 */
38	pthread_mutex_unlock(&thread1Mx);
39	}
40	}
41	
42	/* 두번째 스레드 모듈 */
43	void *runThread2(void *arg)
44	{
45	printf("두번째 스레드 실행\n");
46	/* setCount()함수를 매초마다 호출 */
47	while(1)
48	{
49	/* 계수기가 5를 넘으면 스레드 완료 */
50	if(countNo >= 5) pthread_exit(0);
51	/* thread2 mutex lock */
52	pthread_mutex_lock(&thread2Mx);
53	/* 대기 시작 */
54	pthread_cond_wait(&checker, &thread2Mx);
55	/* setCount()실행후 count번호를 출력 */
56	setCount();
57	printf("두번째 스레드가 얻어온 계수기 번호: %d\n",countNo);
58	/* checker 신호 발생 */
59	pthread_cond_signal(&checker);
60	/* thread2 mutex lock*/
61	pthread_mutex_unlock(&thread2Mx);
62	}
63	}

64	
65	/* 프로세스의 main 함수 */
66	int main()
67	{
68	/* 스레드 선언 */
69	pthread_t thread1_t, thread2_t;
70	
71	/* 첫번째 스레드 생성 */
72	if(pthread_create(&thread1_t, NULL, runThread1, NULL))
73	{
74	printf("첫번째 스레드의 생성에 실패!\n");
75	return 0;
76	}
77	/* 두번째 스레드 생성 */
78	if(pthread_create(&thread2_t, NULL, runThread2, NULL))
79	{
80	printf("두번째 스레드의 생성에 실패!\n");
81	return 0;
82	}
83	
84	/* 스레드들이 작업을 완료할 때까지 대기 */
85	pthread_join(thread1_t, NULL);
86	pthread_join(thread2_t, NULL);
87	
88	/* mutex와 cond 모두 destroy() */
89	pthread_mutex_destroy(&thread1Mx);
90	pthread_mutex_destroy(&thread2Mx);
91	pthread_cond_destroy(&checker);
92	
93	return 1;
94	}

프로그램의 코드작성이 끝났으면 콤파일을 진행하고 실행시켜보자. 프로그램의 실행을 통해 두개의 스레드가 서로 작업을 주고받으며 실행되는것을 확인할수 있다.(그림 4-8)

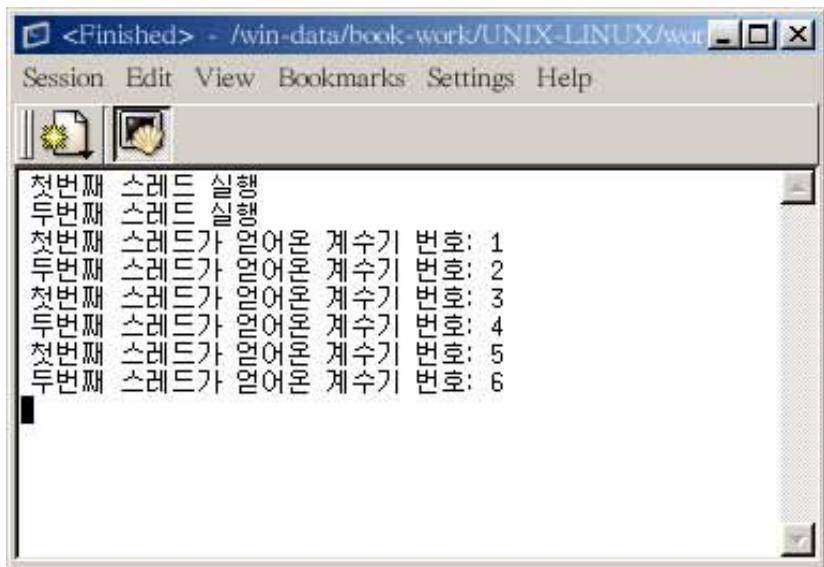



그림 4-8. pth_cond.c의 실행결과

상식

ADSL 이란 무엇인가?

최근 우리 나라에서도 ADSL 회선을 리용한 콤퓨터망가입이 늘어나고있다. 그러면 ADSL 이란 무엇인가?

ADSL 은 asymmetric digital subscriber line 의 략칭으로서 이미 있는 동선으로 된 전화선을 리용하여 고속자료통신을 가능하게 하는 통신수단이다.

음성전송용이나 ISDN(통합봉사수자식망)보다 높은 주파수대역을 리용하여 자료전송을 진행하기때문에 상사식회선의 모뎀에 비하여 수백배에 해당하는 약 10Mbps 이상의 속도로 자료를 전송할수 있다. 그러나 이 통신속도를 실현할수 있는것은 전화국으로부터 가정에로의 아래방향만이며 웃방향에 대해서는 그의 1/10 정도로서 비대칭적이다.

또한 자료통신이 가능한 거리도 수 km 이내로서 짧기때문에 전화국안에 모뎀을 설치하여야 하며 시내망을 가지는 전기통신기관, 기업소가가 봉사를 하지않으면 현실적으로 리용할수 없다.

때문에 현재는 도시 또는 기업소안의 전화망 등에만 리용한다.

ADSL 에는 여러가지 방식이 있다.

현재는 ITU-T(국제전기통신련합 전기통신표준화부문)에서 표준화작업을 진행하는것밖에 인텔회사 등이 각 회사들과 함께 호상접속성을 보장한 일반 ADSL 의 규격을 표준화하려고 하고있다.

제5장

프로쎄스들사이 통신1 (관과 신호기)

서론

이 장에서는 프로쎄스사이의 통신에 대하여 배우도록 하자. 략자로 IPC(Inter Process Communication)로 불리우는 프로쎄스사이의 통신은 여러가지 방법으로 실현하고있다. 이 장에서는 그중에서 관(pipe)과 신호기(semaphore)의 사용에 대해 보기로 한다.

이 장의 첫부분에서는 IPC에 대한 전체적인 소개를 진행하고 그 다음 관과 관의 변형인 이름가진 관(Named pipe)에 대해 배우게 된다. 그리고 관을 리용한 프로그람을 작성하여 보면서 관에 대한 내용을 더 깊이 학습하게 된다.

다음은 신호기에 대해 배우게 된다. 즉 신호기란 무엇이며 신호기에서 사용하는 함수들에는 어떤것들이 있는가 하는것을 학습한다.

마지막으로 신호기와 함께 프로쎄스동기화에 쉽게 적용할수 있는 레코드잠그기에 대하여 취급한다.

5장의 차례를 간단히 소개하면 다음과 같다.

목표

1. 프로쎄스사이의 통신에 대한 개념
2. 관
3. FIFO
4. 신호기
5. 레코드잠그기

제1절. 프로쎄스사이의 통신에 대한 개념

체계프로그람을 만들려고 할때 하나의 프로쎄스로 모든 일을 처리하도록 만들수 있다. 하지만 많은 경우 여러개의 프로쎄스들이 유기적으로 련관되여 동작하면서 체계가 기동되게 만들어야 할 경우가 더 많이 제기된다. 이러한 체계를 만들면서 필수적으로 고려해야 하는것이 바로 프로쎄스사이의 통신인 IPC(Inter Process Communication)이다.

Linux에서 IPC를 구축하기 위해 사용할수 있는 가능한 방법들에는 관, FIFO(named pipe), 신호기, 공유기억기, 통보대기렬 등이 있다. 신호도 IPC의 범주에 속하며 소케트를 리용하여 IPC를 실현할수 있다. IPC를 제대로 실현해야 프로쎄스들사이의 자료흐름이나 기동이 원활하게 수행될수 있으며 밖에서 보았을 때에도 안정된 체계로 인식할수 있다. 우에서 언급된 IPC들에 대해 간단히 설명하면 다음과 같다.

· **관**(Pipe): IPC가운데서 가장 오래된 방법으로서 UNIX의 초창기부터 널리 사용되던 방법이다. 관은 특정한 프로쎄스의 표준출력과 다른 프로쎄스의 표준입력을 서로 련결시키는 방법이다. 쉘에서 많이 사용한다.

· **FIFO**(Named Pipe): 이름가진 관으로 불리우는 FIFO는 일반 관과 동일하게 움직이지만 관과는 다른 부분이 있다. 먼저 이름이 붙여졌다는 말에서 알수 있듯이 파일체계안에는 관작업을 할수 있는 특수한 파일이 존재하고 이를 통해 프로쎄스들이 자료를 공유하는 방법으로 본래의 관이 가지고있던 약점을 퇴치하고있다.

· **신호기**(Semaphore): 신호기는 철도에서 많이 사용되는 용어로서 교차선로상에서 차들의 교통을 보장하는 차단기라는 의미를 가지고있다. IPC에서도 신호기는 여러 프로쎄스가 하나의 자원을 공유할 때 그에 대한 통제권을 관리하는 역할을 하고있다.

· **공유기억기**(Shared Memory): 공유기억기는 토막이라고 불리우는 특정한 기억기령역을 여러개의 프로쎄스가 함께 사용하는것을 의미한다. 즉 기억기의 한 부분을 어떤 프로쎄스가 입력하고 다른 프로쎄스가 그 부분을 가져가는것이라고 볼수 있다. 이때 프로쎄스들은 자료를 자기가 가진 기억기에서 쓰고 읽는다고 볼수 있기때문에 처리속도가 매우 빠르다.

· **통보대기렬**: 프로쎄스가 대기렬속에 통보문을 넣어주면 다른 프로쎄스가 대기렬속에 있는 통보문을 가져가는 개념이 바로 통보대기렬이다. 통보대기렬은 열쇠(key)값이 있으므로 이것을 리용하여 프로쎄스들이 해당 통보대기렬을 인식해서 활용할수 있다.

IPC는 크게 BSD IPC와 System V IPC로 나누어 볼수 있다. 이렇게 갈라 보는것은 판본이나 기능적인 부분보다도 작성자가 누구인가에 따라 나눈것이라고 볼수 있다. 물론 기능이나 내용상측면에서 차이도 있다. BSD IPC는 Berkeley UNIX인 BSD 4.4에서 제공되는 IPC로서 관, 소케트 등이 있다. 그리고 0체계호출함수로 사용할수 있는 read(), write()와 recvmsg(), sendmsg() 등도 제공하고있다. System V IPC는 AT&T에서 개발한 IPC로서 BSD IPC보다는 최신판본이라고 할수 있다. 통보대기렬, 공유기억기, 신호기 등을 제공하고있다. msgget(), msgsnd()등의 체계호출함수를 제공하고있는데 이때 사용되는 완충기나 흐름(stream)은 핵심부를 통해 관리된다.

그러면 이제부터 IPC를 구현하기 위한 방법들에 대해 하나씩 보도록 하자.

상식

AGP(accelerated graphic port)

인텔(Intel)회사가 영상카드전용으로 내놓은 확장모선규격이다.

본래 PCI(주변부분품호상접속)모선의 약 2배에 해당하는 256MB/s의 전송속도를 가지고있었지만 현재의 AGP모선은 그것보다 2배인 512MB/s에 대응하는 2X(2배속)방식을 가지고있는 경우가 많다.

앞으로는 4X(4배속)방식에 의하여 1GB/s를 넘는 고속전송을 예정하고있으며 3차원도형처리프로그람에서 사용하는 영상기억기를 주기억기로부터 떼내여 직접 사용할수 있도록 하고있다.

또한 AGP는 Pentium II를 위하여 개발되였지만 MMX Pentium이나 K6 등을 장비할수 있는 Socket7의 주기판들가운데도 AGP모선에 대응한 제품들이 있다.

제2절. 관

관은 앞에서도 간단히 소개한바와 같이 프로쎄스출력이 다른 프로쎄스의 입력으로 련결되는 단방향(반이중-half duplex)통신통로로 된다. 이것은 한쪽은 주기만 하고 한쪽은 받기만 하는 속성으로부터 나온것이다.

5.2.1. 쉘과 관

관은 쉘에서 흔히 쓰는것으로써 특히 입출력을 재지정할 때 많이 쓴다. 관을 리용하면 두개 또는 그 이상의 프로그람들이 서로의 련관속에서 동작하도록 만들수 있다. 례를 들어 A프로그람의 출력을 B프로그람의 입력으로 넘겨주어야 할 일이 있을 때 관을 리용하게 되면 생각보다 훨씬 간단하게 바라는 목적을 이룩할수 있다. 관은 《|》기호를 리용하게 되는데 간단한 실례를 보면 그림 5-1과 같다.

그림 5-1. 관의 실례

우의 실례를 보면 ps -ef의 결과를 《grep vi》의 입력으로 사용하였다는것을 알수 있다. 여기서 지령의 결과는 《ps -ef》를 통해 현재 실행되고있는 모든 프로쎄스의 정보를 얻어서 《grep vi》를 통해 《vi》라는 단어가 포함된 부분을 찾아낸후 화면에 결과를 현시한것이다. 만일 관을 리용하지 않고 이러한 기능을 수행하는 코드를 작성한다면 그것이 그리 쉽지는 않다.

관을 리용한 간단한 실례를 몇가지 더 보도록 하자.

특정한 포구의 현재 상태를 검사하려고 한다면 다음과 같이 netstat 지령을 사용한 다음 그 결과를 관을 통해 grep에 보낸다. 그리고 grep지령은 원하는 포구번호를 리용하여 관으로 들어온 내용을 검색하게 된다.

```
$ netstat -a | grep 161
```

현재 실행중인 프로쎄스의 개수를 세려고 할 때에도 관을 리용하여 쉽게 구할수 있다. 이때 사용하는 지령으로는 ps와 wc가 있는데 다음과 같이 실행하면 된다.

```
$ ps -ef | wc -l
```

5.2.2. 관체계호출함수

체계에서 새로운 관을 생성하려고 할 때 사용하는 체계호출함수로는 pipe()가 있다. pipe()함수를 리용하여 관을 생성하면 두개의 파일서술자를 얻을수 있다. 이때 얻어진 두개의 파일서술자는 각각 관의 읽기와 쓰기를 위한것이다.

아래에 pipe()함수의 간단한 실례를 주었다.

```
int fileDes[2];
int result;
result = pipe(fileDes);
```

pipe()함수를 실행한 결과가 result인데 만일 result의 값이 -1이면 관생성이 실패한것이다. 이때는 다른 조치를 취하여 관의 실패에 대처해야 한다. 관생성에 성공하면 두개의 파일서술자가 생성된다고 했는데 이때 fileDes[0]은 관으로부터 자료를 입력받기 위한 파일서술자이며 fileDes[1]은 관에 자료를 출력하기 위한 파일서술자이다.

그러면 pipe()함수를 호출하여 관을 생성하는 간단한 실례를 통하여 그의 사용방법을 자세히 보도록 하자.

아래의 실례는 단일프로쎄스안에서 관에 자료를 입력하고 그것을 다시 가져오는것을 보여주는 프로그람이다. 단일프로쎄스가 관을 사용하는것은 전혀 의미가 없는 일이지만 pipe()함수의 사용방법을 쉽게 알수 있으므로 이것을 리용한다.

실례프로그람: ex_pipe.c	
1	#include <stdio.h>
2	
3	/* 완충기 크기 선언 */
4	#define MAXBUF 32
5	
6	int main()
7	{
8	/* 입출력용 완충기를 선언 */
9	char putMsg[MAXBUF], getMsg[MAXBUF];
10	/* 관서술자 선언 */
11	int pipeDes[2];
12	
13	/* 관체계호출 */
14	if(pipe(pipeDes) == -1)
15	{
16	printf("관생성에 실패!");
17	return 0;
18	}
19	/* 관에 넣을 통보문을 작성한후 write()호출 */
20	sprintf(putMsg, "관에 통보문 입출력");
21	printf("INPUT PIPE: %s\n",putMsg);
22	write(pipeDes[1], putMsg, MAXBUF);
23	
24	/* 관에서 통보문을 가져오기 위해 read()호출 */
25	read(pipeDes[0], getMsg, MAXBUF);
26	printf("get msg: %s\n", getMsg);
27	return 1;
28	}

그러면 우의 실례프로그람을 콤파일하여 실행시켜보자. 그림 5-2에 실행결과를 보여주었다. 실행결과를 통하여 단일프로세스안에서 관에 자료를 넣었다가 가져왔다는것을 알 수 있다.

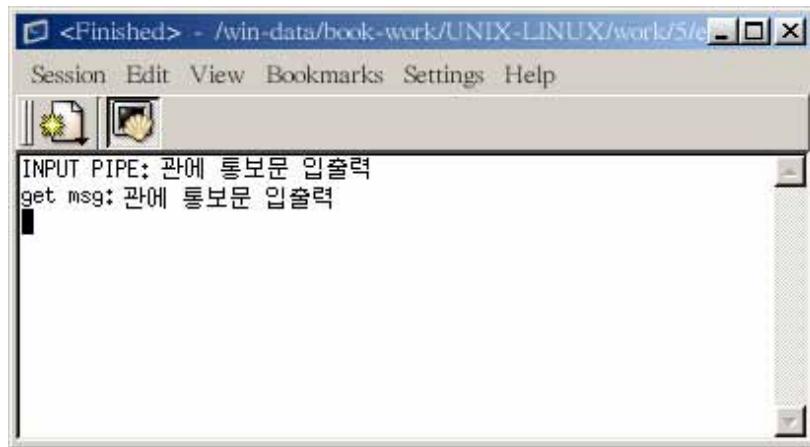



그림 5-2. ex_pipe.c 의 실행결과

관에 입력되는 자료는 FIFO(First In First Out)규칙에 따라 들어온 순서대로 처리된다. 이러한 순서는 사용자가 임의로 변화시킬수 없다. 그리고 입력할수 있는 관의 크기가 무한대가 아니기때문에 범위를 벗어나는 자료가 입력되면 정해진 부분만 입력되고 용량이 초과된 상태에서는 관이 블로크로 되므로 더 이상 자료를 넣을수 없다. 하지만 필요에 따라 관의 상태가 블로크로 되지 않도록 만들어야 할 때가 있다. 례를 들어 이전의 자료는 없어지더라도 최신자료는 관안에 남겨두어야 할 때가 있고 어떤 경우이든지 관에 자료를 읽기 및 쓰기할수 있어야 할 경우도 있다.

이 경우에는 fcntl()함수를 리용하여 관이 블로크화되지 않도록 만들면 된다. fcntl()함수를 실행하여 관의 파일서술자에 O_NDELAY기발이 설정되도록 만들어주면 된다. 이것을 간단히 표기하면 다음과 같다.

```
#include <fcntl.h>
fcntl(fileDes, F_SETFL, O_NDELAY);
```

5.2.3. 부모, 자식프로쎄스사이의 통신

관은 흔히 개별적으로 실행되고있는 프로쎄스보다 부모, 자식프로쎄스사이의 통신에서 많이 활용되고있다. 다시말하여 부모, 자식프로쎄스도 내부자료의 공유는 안되기때문에 프로쎄스통신을 진행해야 하는데 바로 이때 효과적으로 사용할수 있는것이 관이다.

왜냐하면 프로쎄스가 자식프로쎄스를 만들어도 관을 위한 파일서술자정보는 서로 공유하여 사용할수 있기때문이다. 따라서 생성된 관서술자를 리용하여 부모, 자식사이에 통신을 진행하면 특별히 별다른 작업을 하지 않아도 된다는 우점이 있다.

그러면 부모와 자식 프로쎄스가 관을 리용하여 통신하는 실례를 보도록 하자. 이를 위하여 pipe()체계호출함수와 fork()체계호출함수를 사용하도록 한다. 먼저 다음과 같이 pipe()호출에 사용될 배렬을 선언한 다음 pipe()함수를 호출하도록 한다.

```
int pipeDes[2];
pipe(pipeDes);
```

다음 fork()함수를 호출하여 자식프로세스를 생성한다. 이때 fork()함수가 되돌리기(return)하는 PID를 리용하여 부모프로세스와 자식프로세스를 구분하도록 한다. 자식프로세스의 경우에는 관에서 자료를 읽어들인후 화면에 출력하는 작업을 수행한다. 자식프로세스의 작업순서는 다음과 같다.

```
int childPID = fork();
if(childPID == 0)
{
        read(pipeDes[0] , getMsg, MAXBUF);
        printf("GET MSG: %s\n", getMsg);
}
```

부모프로세스는 관에 통보문을 입력시킨다. 이때 입력되는 문장을 사용자로부터 받기 위해 gets()함수를 리용한다. 그리고 만일 입력된 문장이 《quit》이면 실행을 완료하도록 한다. 부모프로세스의 작업순서를 간단히 보면 다음과 같다.

```
if(childPID > 0)
{
        gets(putMsg) ;
        write(pipeDes[1] , putMsg, MAXBUF) ;
        if( !strncmp(putMsg,"quit" ,4)) exit(1) ;
}
```

지금까지 설명한 내용에 기초하여 프로그람을 작성해보도록 하자. 아래에 전체 원천코드를 서술하였다.

실례프로그람: pipe_fork.c	
1	#include <stdio.h>
2	
3	/* 완충기 크기선언 */
4	#define MAXBUF 32
5	
6	int main()
7	{
8	/* 입출력용 완충기를 선언 */
9	char putMsg[MAXBUF], getMsg[MAXBUF];
10	/* 관서술자 선언 */

```
11		int pipeDes[2];
12		/* 자식프로쎄스의 PID용 변수 */
13		int childPID = 0;
14
15		/* 관체계호출 */
16		if(pipe(pipeDes) == -1)
17		{
18			printf("관생성에 실패!");
19			return 0;
20		}
21
22		/* fork()함수 실행, 자식프로쎄스 생성 */
23		childPID = fork();
24
25		/* 자식프로쎄스이면 관에서 통보문을 읽기 */
26		if(childPID == 0)
27		{
28			printf("<<자식프로쎄스>>\n");
29			for(;;)
30			{
31				/* 관에서 통보문을 가져오기 위해 read()호출 */
32				read(pipeDes[0], getMsg, MAXBUF);
33				printf("get msg: %s\n", getMsg);
34				/* 통보문이 quit이면 완료 */
35				if(!strncmp(getMsg, "quit",4))
36				{
37					exit(1);
38				}
39			}
40		}
41		/* 부모프로쎄스이면 관에 통보문을 쓰기 */
42		else if(childPID > 0)
43		{
44			printf("<< 부모프로쎄스 >>\n");
45
46			for(;;)
47			{
48				/* 관에 넣을 통보문을 작성한후 write()호출 */
49				printf("INPUT PIPE: ");
50				gets(putMsg);
51				write(pipeDes[1], putMsg, MAXBUF);
52				/* 통보문이 quit이면 완료 */
53				if(!strncmp(putMsg, "quit",4))
54				{
```

```
55                      exit(1);
56                  }
57                  sleep(1);
58              }
59          }
60          /* 프로쎄스의 생성이 실패인 경우 */
61          else
62          {
63              printf("프로쎄스의 생성에 실패했습니다.\n");
64          }
65          return 0;
66  }
```

우의 프로그람을 콤파일하여 실행시켜보자.

실행결과는 그림 5-3과 같다. 프로그람의 실행과정을 통하여 부모프로쎄스에서 입력한 통보문을 자식프로쎄스에서 관을 리용하여 가져간다는것을 알수 있다. 그리고 quit통보문을 리용하여 모든 프로쎄스를 완료하였다는것을 확인할수 있다.

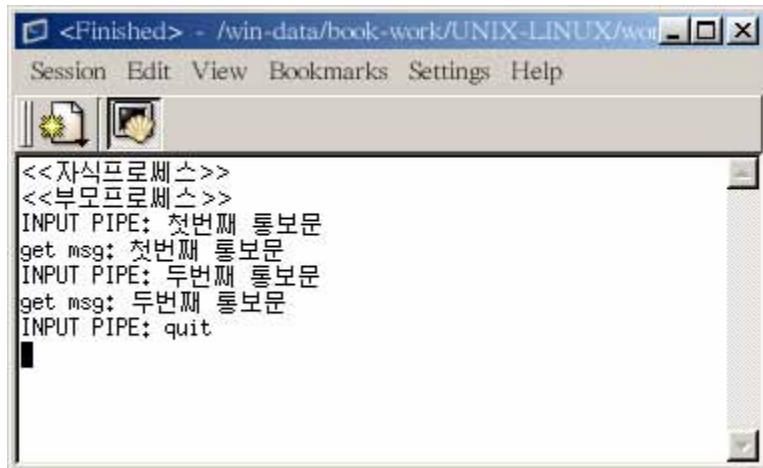



그림 5-3. pipe_fork.c 의 실행결과

5.2.4. 관의 신호처리

관의 실행과정에 자체로 신호가 발생할수 있다. 이러한 경우 해당 신호에 대한 처리를 진행하여 관의 사용으로 인한 비정상탈퇴를 막도록 대책을 세워야 한다. 관에서의 신호처리는 일반적인 신호처리와 같다. 다시말하여 signal()함수를 리용하여 신호조종기를 등록하고 실행시키려는 함수를 작성하면 된다.

관에서 신호가 발생하는 대표적인 경우로는 다음과 같은것이 있다. 관에 쓰려고 하는 프로쎄스가 있는데 읽으려는 프로쎄스가 탈퇴 등으로 하여 사라지면 핵심부는 SIGPIPE 신호를 쓰려고 하는 프로쎄스에 전송한다. 이를 통하여 관에 자료를 쓰기하면 문제가 생기게 된다는것을 알리게 된다.

그러면 이번에는 관신호을 발생시키고 이것을 처리하는 실례프로그람을 통하여 관의 신호처리방법에 대하여 보도록 하자. 작성할 실례에서는 먼저 관신호를 처리할 함수를 작성하고 signal함수를 통하여 조종기를 등록하게 된다.

```
int sigpipeHandler( );
signal(SIGPIPE, sigpipeHandler);
```

다음 pipe()함수를 리용하여 관을 만들고 이것을 사용할 파일서술자를 설정한다. 그리고 이전에 취급했던 실례에서와 같이 자식프로쎄스를 만들고 통보문을 서로 주고받는다. 이때 《quit》라는 문장을 입력받으면 자식프로쎄스는 탈퇴한다. 하지만 부모프로쎄스는 탈퇴하지 않고 문장을 계속 쓰기하도록 만든다.

그러면 자식프로쎄스는 이미 탈퇴되였기때문에 읽기할 프로쎄스가 없는 상태에서 write가 이루어지는것이므로 부모프로쎄스는 핵심부로부터 SIGPIPE신호을 받게 된다. 이렇게 되면 등록된 신호조종기가 호출되고 관신호를 처리하게 된다.

이러한 과정을 수행하는 전체 프로그람의 원천코드는 다음과 같이 작성할수 있다.

실례프로그람: pipe_signal.c	
1	#include <stdio.h>
2	#include <signal.h>
3	
4	/* 완충기 크기선언 */
5	#define MAXBUF 32
6	
7	/* sigpipe신호를 처리할 조종기 */
8	int sigpipeHandler()
9	{
10	/* 필요한 작업을 처리한후 프로그람완료 */
11	printf("\n 관이 비정상적으로 닫겼습니다.\n");
12	printf("sigpipe조종기 호출, 작업완료중… \n");
13	sleep(1);
14	printf("\n<<<작업완료>>>\n");
15	exit(1);
16	}
17	
18	int main()
19	{
20	/* 입출력용완충기를 선언 */

21	char putMsg[MAXBUF], getMsg[MAXBUF];
22	/* 관서술자선언 */
23	int pipeDes[2];
24	/* 자식프로쎄스의 PID용변수 */
25	int childPID = 0;
26	
27	/* sigpipe조종기를 등록, signal()함수사용 */
28	printf("sigpipe조종기설정\n\n");
29	signal(SIGPIPE, sigpipeHandler);
30	
31	/* 관체계호출 */
32	if(pipe(pipeDes) == -1)
33	{
34	printf("관생성에 실패!");
35	return 0;
36	}
37	
38	/* fork함수 실행, 자식프로쎄스 생성 */
39	childPID = fork();
40	
41	/* 자식프로쎄스이면 관에서 통보문을 읽기 */
42	if(childPID == 0)
43	{
44	printf("<<자식프로쎄스>>\n");
45	
46	for(;;)
47	{
48	/* 관에서 통보문을 가져오기 위해 read()호출 */
49	read(pipeDes[0], getMsg, MAXBUF);
50	printf("GET MSG: %s\n", getMsg);
51	/* 통보문이 quit이면 완료 */
52	if(!strncmp(getMsg, "quit",4))
53	{
54	exit(1);
55	}
56	}

```
57          }
58          /* 부모프로쎄스이면 관에 통보문을 쓰기 */
59          else if(childPID > 0)
60          {
61                  printf("<<부모프로쎄스>>\n");
62
63                  for(;;)
64                  {
65                          /* 관에 넣을 통보문을 작성한후 write()를 호출 */
66                          printf("INPUT PIPE:");
67                          gets(putMsg);
68                          /* 통보문이 quit이면 읽기용 관 닫기 */
69                          if(!strncmp(putMsg, "quit",4))
70                          {
71                                  close(pipeDes[0]);
72                                  /* 관에 쓰기를 하고 1s 기다린다. */
73                                  write(pipeDes[1], putMsg, MAXBUF);
74                                  sleep(1);
75                          }
76                          /*  */
77                          write(pipeDes[1], putMsg, MAXBUF);
78                          sleep(1);
79                  }
80          }
81          /* 프로쎄스의 생성이 실패인 경우 */
82          else
83          {
84                  printf("프로쎄스의 생성에 실패했습니다.\n");
85          }
86          return 0;
87  }
```

우와 같이 프로그람을 작성하였으면 실행을 시켜보자.

그림 5-4는 프로그람을 실행시키고 발생한 신호를 처리한 결과를 보여주고있다.

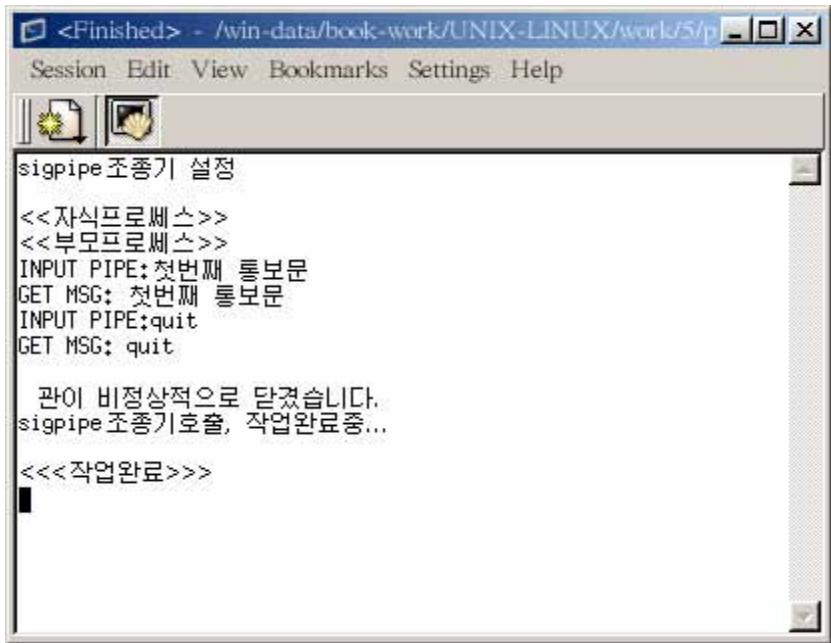



그림 5-4. pipe_signal.c의 실행결과

5.2.5. 관과 exec 함수

쉘상에서 관을 리용하여 서로의 결과를 주고받는것을 이미 보았는데 이것을 pipe() 체계호출함수를 리용한 프로그람에서도 활용할수 있다. 이때 사용되는 체계호출함수가 바로 exec()이다.

리해를 쉽게 하기 위해 exec()함수를 활용한 실례를 먼저 보기로 하자. 아래의 실례는 쉘지령인 ps를 실행시킨것으로써 execvp()함수를 리용한다. execvp는 exec계렬의 함수로서 지령을 위한 인수들과 지령이름을 리용한다.

```
/* exec()를 리용하여 ps -ef 를 실행시킨 실례 */
main()
{
      char *ps[3] = {"ps", "-ef", NULL};
      execvp(ps[0], ps);
      exit(0)
}
```

그러면 exec프로쎄스를 실행하고 그 결과를 관을 통해 다른 exec프로쎄스에 전달하려면 어떻게 해야 하겠는가? 이 문제는 간단히 말하여 "ps -ef | grep sys"와 같은 지령을 exec프로쎄스와 pipe를 리용하여 수행하려면 어떻게 해야 하는가 하는 문제를 해결하는것이다.

이러한 지령을 수행하기 위하여 다음과 같은 배렬의 지적자를 리용하여 변수를 설정한다.

```
char *ps[3] = {"ps" , "-ef" , NULL};
char *grep[3] = {"grep" , "sys" , NULL};
```

그 다음 pipe()함수를 리용하여 관에서 사용할 파일서술자를 얻도록 한다. 먼저 실행되는 자식프로세스에 dup2()지령을 리용하여 본래의 파일서술자를 대신할 새로운 파일서술자를 만들도록 한다. 이때 대신 리용되는 파일서술자는 관에 자료를 쓰기하는 파일서술자가 된다. 이렇게 한 다음 execvp()를 리용하여 미리 설정해두었던 ps명령을 실행시킨다.

```
dup2 ( pipeDes [1] , 1);
close ( pipeDes [0] );
close ( pipeDes [1] );
execvp ( ps [0] , ps);
```

이번에는 자식프로세스가 실행한 ps지령의 결과를 관으로 받아서 처리할 부모프로세스에 대하여 보기로 하자. 부모프로세스도 dup2()지령을 리용하여 관을 읽는 파일서술자를 새로운 파일서술자로 대치시킨다. 그 다음 미리 설정해 둔 grep를 execvp()함수를 리용하여 실행시킨다.

```
dup2 ( pipeDes [0] , 0);
close ( pipeDes [0] );
close ( pipeDes [1] );
execvp ( grep [0] , grep );
```

그러면 지금까지 설명한 내용에 기초하여 exec프로세스를 실행하고 그 결과를 관을 통해 다른 exec프로세스에 전달하는 실례프로그람의 원천코드를 보기로 하자.

실례프로그람: pipe_exec.c	
1	#include <stdio.h>
2	
3	int main()
4	{
5	/* 관서술자선언 */
6	int pipeDes[2];
7	
8	/* 자식프로세스의 PID용 변수 */
9	int childPID = 0;
10	

11	/* 실행할 쉘지령을 정의 */
12	char *ps[3] = {"ps", "-ef", NULL};
13	char *grep[3] = {"grep", "sys", NULL};
14	
15	/* 관체계호출 */
16	if(pipe(pipeDes) == -1)
17	{
18	printf("관생성에 실패!");
19	return 0;
20	}
21	
22	/* fork()함수 실행, 자식프로쎄스의 생성 */
23	childPID = fork();
24	
25	/* 자식프로쎄스이면 ps지령을 실행 */
26	if(childPID == 0)
27	{
28	printf("<<자식프로쎄스: ps실행 >>\n");
29	dup2(pipeDes[1], 1);
30	close(pipeDes[0]);
31	close(pipeDes[1]);
32	execvp(ps[0], ps);
33	printf("ps 실행에 실패!\n");
34	}
35	/* 부모프로쎄스이면 grep지령을 실행 */
36	else if(childPID > 0)
37	{
38	printf("<<부모프로쎄스: ps결과를 받아 grep실행>>\n");
39	dup2(pipeDes[0],0);
40	close(pipeDes[0]);
41	close(pipeDes[1]);
42	execvp(grep[0],grep);
43	printf("grep의 실행에 실패\n");
44	return;
45	}
46	/* 프로쎄스의 생성에 실패한 경우 */

47	else
48	{
49	printf("프로쎄스의 생성에 실패하였습니다.\n");
50	}
51	return 0;
52	}

우의 프로그람을 실행시키면 ps -ef지령을 리용하여 실행중인 프로쎄스들을 찾은 다음 grep sys지령을 리용하여 sys라는 문구를 가진 프로쎄스를 화면에 출력하게 된다. 그림 5-5에 그 결과를 보여주었다.

그림 5-5. pipe_exec.c 의 실행결과

상식

데이터그램(Datagram)

콤퓨터망에서 자료통신용 규약을 매개의 통보문에 포함시켜 전송하는 파케트교환방식의 일종으로서 송신말단에서 수신말단까지의 경로를 정하기 위한 정보를 내부에 가지고있는 독립적인 파케트방식이다.

제3절. FIFO

지금까지 서술한 관은 프로세스사이의 통신에서는 아주 편리한 기능을 보장하고있지만 몇가지 문제점을 가지고있다.

첫째로 부모-자식프로세스사이의 통신에서는 뛰여난 능력을 발휘할수 있지만 부모와 자식의 관계가 아닌 다른 프로세스와의 통신에서는 크게 쓸모가 없다는 약점이 있다.

둘째로 관은 프로세스들의 수행이 끝난 다음에는 없어진다는것이다. 이런 문제점으로 하여 관을 서로 다른 프로세스사이의 통신에서는 리용하기가 힘들다.

이러한 문제점을 극복하기 위한 방도로서 변형된 관이 개발되였는데 이것이 바로 이름가진 관(named pipe)으로 불리우는 FIFO이다. FIFO는 파일처럼 이름을 가지며 관리할수 있다. 즉 관기능을 담당할 파일이 생성되여 지우기전까지 계속 파일체계에 남아있게 된다. 또한 접근권한을 리용하여 관의 접근을 제한할수도 있다.

쉘상에서 FIFO를 만들려면 mknod지령을 p추가선택항목과 함께 사용하면 된다.

반드시 명심해야 할것은 FIFO는 일반파일이 아닌 특수파일의 일종이라는것이다.

그러면 FIFO를 리용하여 프로그람을 작성하는 방법을 보도록 하자. 일반적으로 FIFO와 관을 사용하는데는 특별한 차이가 없다. 하지만 FIFO는 파일로 존재하기때문에 생성과 열기(open)에서 차이점을 가지고있다. FIFO를 생성하기 위해서는 mkfifo()함수를 리용하여야 한다. mkfifo()와 함께 파일이름과 접근권한을 지적해주면 FIFO가 생성된다. 례를 들면 다음과 같다.

```
mkfifo("FIFO", 0666);
```

mkfifo()함수호출에 실패하면 -1이 되돌려진다. 만일 이미 FIFO가 존재한다면 생성에는 실패하지만 이전의 FIFO를 그냥 사용할수 있다. FIFO를 생성했으면 open()지령을 리용하여 FIFO의 파일서술자를 얻어오도록 한다. 이것이 본래의 관과 가장 큰 차이점이라고 할수 있다. 실례를 들어보면 다음과 같다.

```
int fileDes;
fileDes = open( "FIFO", O_RDWR);
fileDes = open( "FIFO", O_RDONLY);
fileDes = open( "FIFO", O_WRONLY);
```

만일 FIFO를 열기하면서 블로크로 되지 않도록 하려면 O_NDELAY기발을 함께 사용하면 된다. 읽기전용으로 FIFO를 열기하면서 블로크로 되지 않도록 만들려면 아래의 사용형식과 같이 하여야 한다.

```
fileDes = open( "FIFO", O_RDONLY | O_NDELAY);
```

그러면 지금까지 소개한 내용을 바탕으로 하여 FIFO를 만들고 이것을 리용하여 통신을 진행하는 실례프로그람을 만들어보자. 이때 사용되는 프로쎄스는 부모, 자식관계를 가지지 않는 전혀 별개의 프로쎄스가 되여야 한다.

먼저 FIFO를 리용하여 통보문을 수신할 프로쎄스의 작업에 대하여 보자.

FIFO를 만들거나 리용하기 위해 fcntl.h머리부파일을 불러들인다. 그리고 수신할 통보문의 크기와 완충기를 선언해 준다. 그런 다음 FIFO를 생성하도록 한다.

```
#include <fcntl.h>
#define MAXBUF 32
char buffer[MAXBUF];
mkfifo( "FIFO" , 0666);
```

FIFO의 생성이 끝났으면 open을 통해 FIFO를 열도록 한다. 모든 작업이 원만히 되였으면 read를 리용하여 FIFO로부터 buffer안에 자료를 입력한다. 만일 읽어들인 자료가 quit로 시작되면 프로쎄스를 탈퇴시킨다.

```
int fileDes;
fileDes = open( "FIFO" , O_RDWR);
read(fileDes, buffer, MAXBUF);
if( !strncmp(buffer,"quit" ,4) ) exit(0);
```

이번에는 FIFO에 자료를 입력하는 프로쎄스의 작업을 보도록 하자. 이 프로쎄스는 FIFO를 쓰기전용으로 열기한후 사용자로부터 전송할 통보문을 입력받는다. 그 다음 write 함수를 리용하여 통보문을 FIFO에 넣는다.

```
int fileDes;
fileDes = open( "FIFO" , O_WRONLY);
gets(buffer);
write(fileDes, buffer, MAXBUF);
```

지금까지 소개한 내용을 기초로 하여 전체 과정에 대한 원천코드를 작성하여 보자.

아래의것은 통보문을 읽기하는 프로그람의 원천코드이다.

실례프로그람: rcvFifo.c	
1	#include <stdio.h>
2	#include <fcntl.h>
3	#include <errno.h>
4	
5	/* 통보문의 크기선언 */
6	#define MAXBUF 32
7	

8	int main()
9	{
10	/* 파일서술자와 완충기선언 */
11	int fileDes;
12	char buffer[MAXBUF];
13	
14	/* mkfifo()를 리용하여 FIFO 생성 */
15	if(mkfifo("FIFO", 0666) == -1)
16	{
17	if(errno != EEXIST)
18	printf("FIFO 관생성에 실패");
19	}
20	
21	/* open()을 리용하여 FIFO 열기 */
22	fileDes = open("FIFO", O_RDWR);
23	if(fileDes < 0)
24	{
25	printf("FIFO 관열기에 실패");
26	return 0;
27	}
28	
29	/* 통보문을 계속 읽어들인다. */
30	for(;;)
31	{
32	/* FIFO로부터 통보문을 읽기 */
33	if((read(fileDes, buffer, MAXBUF)) <0)
34	{
35	printf("통보문의 읽기에 실패");
36	break;
37	}
38	else
39	{
40	/* 읽은 통보문을 화면에 출력 */
41	printf("읽은 통보문: %s\n", buffer);
42	/* 통보문이 quit이면 완료 */
43	if(!strncmp(buffer, "quit", 4))
44	{
45	exit(0);
46	}
47	}
48	}
49	return 1;
50	}

다음은 통보문을 FIFO에 쓰는 프로그람의 원천코드이다.

실례프로그람: sendFifo.c

```
1   #include <fcntl.h>
2   #include <stdio.h>
3
4   /* 통보문의 크기선언 */
5   #define MAXBUF 32
6
7   int main()
8   {
9           /* 파일서술자와 완충기선언 */
10          int fileDes;
11          char buffer[MAXBUF];
12
13          /* open()을 리용하여 FIFO열기 */
14          fileDes = open("FIFO", O_WRONLY);
15          if(fileDes < 0)
16          {
17                  printf("FIFO 관의 열기에 실패");
18                  return 0;
19          }
20
21          /* 통보문을 계속 전송한다. */
22          for(;;)
23          {
24                  /* 사용자로부터 통보문입력받기 */
25                  printf("통보문입력: ");
26                  gets(buffer);
27                  /* 입력받은 통보문을 FIFO에 쓴다. */
28                  if(write(fileDes, buffer, MAXBUF) == -1)
29                  {
30                          printf("관에 대한 통보문의 쓰기에 실패");
31                          break;
32                  }
33                  /* 통보문이 quit이면 완료 */
34                  if(!strncmp(buffer, "quit", 4))
35                  {
36                          exit(0);
37                  }
38          }
39          return 1;
40  }
```

코드작성이 끝나면 두개의 쉘창(또는 서로 떨어진 두대의 콤퓨터)을 리용하여 프로그람들을 실행시켜 보자. 먼저 그림 5-6과 같이 rcvFifo프로그람을 실행시킨다. 그리고 다른 창을 리용하여 sendFifo프로그람을 실행시킨다.(그림 5-7)

그림 5-6. rcvFifo.c의 실행결과

그림 5-7. sendFifo.c의 실행결과

이때 rcvFifo지령을 실행시킨 다음 해당 등록부내부를 보면 FIFO특수파일이 생성된것을 볼수 있다.

콤파일러(compiler)와 해석기(interpreter)는 어떤 차이가 있는가?

콤파일러란 원리적으로 볼 때 하나의 프로그람작성언어를 다른 종류의 언어로 변환하는 프로그람을 말한다. 그러나 일반적으로 사용되는 의미는 고급프로그람작성언어로 작성된 원시프로그람을 콤퓨터가 직접 해득할수 있는 대상코드(object code)로 번역하는 프로그람을 말한다.

이와는 달리 해석기는 자동프로그람작성기의 일종으로서 원시프로그람에서 한개 문장을 받으면 번역하여 즉시 실행으로 옮기고 이어서 다음 원시문을 번역하는 형식으로 처리를 진행하는 프로그람이다.

제4절. 신호기

앞에서 스레드에 대하여 설명하면서 스레드동기화에 대한 문제를 취급하고 mutex와 스레드조건변수를 활용하여 문제를 해결하는 방법을 보았다. 스레드에서 동일한 령역이나 자원을 동시에 리용하면 문제가 발생하는것처럼 프로쎄스들사이에서도 같은 원인으로 하여 문제가 발생할수 있다. 그러므로 이 절에서는 프로쎄스들사이에서 발생할수 있는 동기화문제를 해결하는 방법에 대해 설명하려고 한다.

5.4.1. 신호기 소개

스레드에서 mutex를 리용하여 스레드사이의 동기화문제를 해결했다면 프로쎄스에서는 신호기를 리용하여 이 문제를 해결한다. 신호기는 프로쎄스들사이에 리용되는 자료 등의 자원을 보호하는데 목적을 둔 프로쎄스들사이의 통신(IPC)방법이다. 하지만 신호기는 관이나 통보대기렬과 같은 자료전송을 목적으로 하는 일반적인 프로쎄스들사이 통신과는 다르다.

프로쎄스들사이에서 동시에 특정한 자료에 접근해야 할 때 특히 서로 같은 시간에 접근할 때에는 하나의 프로쎄스만 접근하도록 만들어주어야 한다. 이것은 스레드에서 mutex를 사용해야 하는 원리와 같다. 그러면 신호기의 동작원리에 대해 간단히 살펴보자.

먼저 공유된 자원에 대하여 신호기를 생성하여 신호기가 자원을 가리키도록 한다. 이때 신호기안에는 해당 자원에 접근이 허용된 프로쎄스의 개수가 나타나게 된다. 례를 들어 이 값이 0이면 자원에 접근할수 있는 프로쎄스가 없다는것을 의미한다.

따라서 자원에 접근하려고 하는 프로쎄스들은 이 값을 검사하면 된다. 만일 이 값이 0보다 크면 자원에 접근할수 있다. 자원에 접근할 때에는 신호기값을 하나 감소시키고 작업이 끝나면 다시 하나 높여주면 된다. 만일 신호기값이 1이라면 최대 하나의 프로쎄스만 해당 자원을 사용할수 있게 된다.

왜냐하면 어떤 프로쎄스가 자원을 사용할 때에는 0이 되기때문에 아무도 접근할수 없으며 작업이 끝나면 자원에 대한 해제를 하여야 또 다른 프로쎄스가 작업을 진행할수 있기때문이다.

이러한 원리를 신호기에 구현하려면 먼저 신호기를 초기화하는 과정이 필요하며 신호기값을 알아보는 과정도 필요하다. 그리고 정황에 맞게 신호기값을 증가 또는 감소시키는 과정이 필요하다.

Linux에서는 이러한 신호기를 적용할수 있는 구조(struct)와 체계호출함수들을 제공하고있다.

5.4.2. 신호기체계호출함수

체계준위에서 신호기를 제공하기 위하여 핵심부에서는 신호기를 위한 구조(구조체)를 리용하여 신호기를 관리하게 된다. 이때 사용하는 구조는 semid_ds인데 일반적으로 다

음과 같이 구성되여있다.

```
struct semid_ds
{
        struct ipc_perm sem_perm;
        unsigned long int sem_nsems;
        _time_t sem_otime;
        _time_t sem_ctime;
        unsigned long int _unused1;
        unsigned long int _unused2;
        unsigned long int _unused3;
        unsigned long int _unused4;
};
```

제일 먼저 나오는 perm값은 신호기에 대한 접근권한을 의미하는것으로서 일반파일에서와 같다. nsems는 생성하려고 하는 신호기의 크기를 의미한다. 그리고 otime은 신호기와 관련된 작업을 수행하는 마지막 시간을 의미하며 ctime은 구조의 자료들이 갱신된 마지막 시간을 의미한다.

Linux에서는 구조와 신호기를 리용한 기본적인 작업을 위하여 체계호출함수로 다음과 같은것들을 제공하고있다.

```
int semget();
int semop();
int semctl();
```

그러면 이러한 체계호출함수에 대하여 하나씩 보도록 하자. 먼저 semget()에 대해 보면 이 함수는 신호기를 새로 생성하거나 또는 이미 만들어진 신호기를 얻기 위해 사용하는 함수이다. semget()함수의 사용형식을 보면 다음과 같다.

```
int semget(key_t key, int nsems, int semflg);
```

semget()함수에서 제일 먼저 사용된 key인수는 신호기의 열쇠값으로서 다른 신호기와 구별하여 리용하는 ID로 된다. 신호기가 생성되면 key값을 리용하여 신호기에 접근할수 있다. 그리고 두번째로 사용된 nsems인수는 사용하려는 신호기의 개수를 의미한다.

보통 1로 사용되는 이 값은 신호기를 생성할 때에 꼭 필요하다. 신호기를 생성하지 않고 그냥 신호기의 얻기만을 목적으로 할 때에는 0을 사용해도 된다. 세번째로 사용된 semflg 인수는 신호기에 접근할 때 사용하는 기발로 된다. 이 기발을 리용하여 신호기의 생성과 접근권한을 가질수 있다. 기발로 사용할수 있는 선언자에는 다음과 같은것들이 있다.

- IPC_CREAT: 지정된 Key값을 리용하여 신호기를 생성한다.

• IPC_EXCL: IPC_CREAT와 함께 사용한다. 만일 Key를 가진 신호기가 이미 존재하면 오유를 되돌려준다.

• 접근권한지정: 일반파일에서 접근권한을 지정하듯이 수자의 묶음을 사용한다. (례: 666)

이러한 기발들은 《|》를 리용하여 련속적으로 지정할수도 있다. 례를 들어 《0666 | IPC_CREAT》처럼 사용할수 있다. 만일 신호기를 생성하기 위하여 IPC_CREAT만을 지정한 경우에는 동일한 열쇠값을 가진 신호기가 이미 있으면 본래 신호기의 ID를 되돌려 준다.

만일 열쇠값을 7654로 사용하고 접근권한을 666으로 설정하여 신호기를 새롭게 생성하려면 다음과 같이 하면 된다.

```
int Sem_id = semget((key_t)7654, 1, 0666 | IPC_CREAT);
```

이번에는 semop()체계호출함수에 대해 살펴보자. semop()함수는 신호기연산을 실행하는 함수로서 사용형식은 다음과 같다.

```
int semop(int sem_id, struct sembuf *ops, num);
```

여기서 첫번째인수인 sem_id는 semget()에서 얻어온 신호기의 ID가 된다. 그리고 두번째인수인 sembuf형의 지적자형변수 ops는 신호기에 대하여 수행하려고 하는 연산을 지정한 sembuf형의 배렬에 대한 지적자가 된다.

이때 sembuf구조(struct)는 다음과 같이 구성되여있다.

```
struct sembuf
{
        short sem_num;
        short sem_op;
        short sem_flg;
};
```

여기서 sem_num은 신호기의 번호를 의미한다. 만일 신호기의 원소가 하나라면 0이 된다. sem_op에는 신호기값을 변경시키기 위한 값이 들어있다. 마지막으로 sem_flg는 신호기의 기발을 설정하는데 사용되는 값으로서 일반적으로 SEM_UNDO가 사용된다. SEM_UNDO로 설정하면 신호기에 대한 변경을 관리하며 프로쎄스가 탈퇴할 때 특별히 신호기를 제거하지 않아도 체계가 자동적으로 신호기를 해제시켜준다.

이번에는 semctl()체계호출함수에 대해 보기로 하자. semctl()함수는 신호기를 직접 조종하는데 사용되는 함수로서 다음과 같은 사용형식을 가진다.

```
int semctl(int sem_id, int semnum, int cmd, union semun arg);
```

semctl()함수에서 사용되는 첫번째 인수인 sem_id는 semget()를 통해 얻어온 신호기의 ID로 되고 두번째 인수인 semnum은 신호기의 번호를 의미한다. 세번째 인수인 cmd는 신호기

가 수행할 지령으로 된다. 이때 사용할수 있는 선언자로는 다음과 같은것들이 있다.

- IPC_STAT: 신호기의 상태를 마지막 인수인 arg에 보관
- IPC_SET: 마지막 인수인 arg의 semid_ds구조체가 가지고있는 내용을 기초로 신호기의 접근권한을 변경
- IPC_RMID: 신호기 삭제
- GETALL: 모든 신호기의 값을 얻은 다음 마지막 인수인 arg의 정수배렬에 보관
- GETCNT: 자원의 접근을 기다리고있는 프로세스의 수를 되돌리기
- GETPID: 마지막으로 semop()함수를 실행한 프로세스의 PID를 되돌리기
- SETVAL: 신호기를 마지막 인수인 arg의 val값으로 설정
- SETALL: 모든 신호기의 값을 마지막 인수인 arg의 값으로 설정

마지막 인수로 사용된 union semun형의 arg에 대하여 보자. 먼저 semun형을 보면 다음과 같다.

```
union semun {
        int val;
        struct semid_ds *buf;
        unsigned short *array;
        struct seminfo *infobuf;
        void *pad;
};
```

semun은 union형이기때문에 인수로 사용된 arg는 semun형중에서 하나의 값으로 사용된다. 이것은 방금전에 살펴보았던 cmd인수와 련관이 있다. 다시말하여 semun의 첫번째 형인 val은 SETVAL을 위한 값으로 활용된다. 그리고 semid_ds*형의 buf는 IPC_STAT와 IPC_SET를 위한 완충기로 사용된다. array의 경우에는 GETALL 지령과 SETALL지령을 위한 배렬로 된다. 마지막으로 infobuf는 IPC_INFO를 위한 완충기로 사용된다.

지금까지 신호기의 체계호출함수들에 대해 하나씩 살펴보았다. 체계호출함수들에 대한 설명가운데서 잘 리해가 되지 않는 부분들도 있겠지만 실례프로그람들을 리용하여 실지로 함수를 사용하여 보면 쉽게 리해할수 있다.

5.4.3. 신호기를 리용한 실례프로그람

여기서 설명할 실례프로그람은 파일에 통보문을 입력하는 프로그람이다. 이때 신호기를 리용하여 파일에 통보문을 입력하는 프로세스는 오직 하나가 되도록 만든다. 그리고 시험을 위하여 하나의 프로세스는 신호기의 값을 증가만 시키고 또 다른 프로세스는 신호기의 값을 감소만 시키도록 만든다.

신호기의 값을 증가만 시키는 프로세스의 경우에는 다른 프로세스의 도움이 없이도 작업을 수행할수 있다. 하지만 신호기의 값을 감소만 시키는 프로세스의 경우에는 다른 프로세스가 신호기의 값을 증가시켜주지 않으면 실행을 계속할수가 없다. 왜냐하면 신호기의 값이 0이 되는 순간부터 1이 될 때까지 계속 대기해야 하기때문이다.

그럼 먼저 신호기의 값을 증가시키는 plusSem프로그람을 작성해보자. 먼저 semget() 함수를 리용하여 신호기의 ID를 얻어오도록 한다. 만일 신호기가 생성되여있지 않으면 그것을 생성시키고 sembuf구조형의 변수를 선언한 다음 이것을 초기화한다.

```
  semId  =  semget((key_t)1234,  1,  0666  |  IPC_CREAT);/*
semget()실행*/
  struct sembuf semB;
  semB.sem_flg = SEM_UNDO;
  semB_num = 0;
```

또한 semctl()함수를 리용하여 신호기의 값을 초기화하고 파일에 통보문을 입력한 다음 신호기의 값을 증가시킨다. 정상적인 프로그람이라면 파일에 통보문을 입력하기전에 신호기의 값을 감소시켜주어야 한다.

```
semctl(semId, 0, SETVAL, 1); /* 신호기의 초기화 */

file = fopen( “./db.txt” , “a+” ); /* 파일작업수행 */
fprintf(file, “plusSem 프로쎄스통보문 보관\n” );
fclose(file);

semB.sem_op = 1; /* 신호기의 값증가 */
semop(semId, &semB, 1);
```

신호기를 리용한 작업이 끝났으면 아래와 같이 semctl()함수를 리용하여 신호기를 제거한다.

```
semctl(semId, 0, IPC_RMID, 0);
```

이때 시험을 위해 아래와 같이 신호기의 실행중간에 신호기를 수정한 프로쎄스의 PID와 자기자신의 PID를 출력하는 모듈을 삽입하도록 하자.

```
/* 신호기에 마지막으로 수정을 가한 프로쎄스의 PID 를 출력*/
int proId = semctl(semId, 0, GETPID, 0);

/* 자기자신의 PID 출력*/
printf ("minus 신호기의 PID: %d\n", getpid());
```

이번에는 신호기의 값을 감소만 시키는 minusSem프로그람을 작성해보자. 먼저 semget() 함수를 리용하여 신호기의 ID를 구하도록 한다. 그 다음 신호기의 값을 감소시키는 작업과 파

일에 통보문을 입력하는 작업을 반복해서 수행한다.

```
semId = semget((key_t)1234, 1, 0666 | IPC_CREAT);
for(...;...;...)
{
     /*신호기의 값을 감소시킨다.*/
     semB.sem_op = -1;
     semoop(semId, &semB, 1);
     /* 아래부분은 생략 */
}
```

minusSem프로세스가 실행하는 경우 plusSem프로세스의 도움이 없이는 작업을 끝까지 수행할수 없다. 그것은 자기의 능력으로는 0이 된 신호기의 값을 증가시킬수 없기때문이다. 그러면 지금까지 설명한 내용에 기초하여 작성한 전체 원천코드를 보도록 하자. 먼저 plusSem프로그람의 원천코드는 다음과 같다.

실례프로그람: plusSem.c

```
1   #include <stdio.h>
2   #include <unistd.h>
3   #include <stdlib.h>
4
5   /* 신호기를 위한 머리부파일들 */
6   #include <sys/types.h>
7   #include <sys/ipc.h>
8   #include <sys/sem.h>
9
10  /* 신호기의 값을 증가시키는 main함수 */
11  int main(int argc, char *argv[ ])
12  {
13          /* 신호기와 파일을 위한 변수선언 */
14          int step;
15          int semId, proId;
16          FILE *file;
17          struct sembuf semB;
18
19          /* sembuf의 초기값설정 */
20          semB.sem_flg = SEM_UNDO;
21          semB.sem_num =0;
22
23          /* semget()를 리용하여 신호기의 ID 구하기 */
```

```
24          semId = semget((key_t)1234, 1, 0666 | IPC_CREAT);
25
26          /* 신호기의 초기값설정 */
27          if(semctl(semId, 0, SETVAL, 1) == -1)
28          {
29                  fprintf(stderr, "신호기의 초기화에 실패하였습니다.\n");
30                  exit(0);
31          }
32
33          /* plusSem 프로쎄스의 PID 출력 */
34          printf("PLUS 신호기의 PID : %d\n", getpid());
35
36          /* 파일에 통보문을 5번 기입한후 완료 */
37          for(step = 0; step < 5; step++)
38          {
39                  /* 신호기에 마지막으로 수정을 가한 프로쎄스의 PID 출력 */
40                  proId = semctl(semId, 0, GETPID, 0);
41                  printf("신호기를 변경한 마지막 PID: %d\n",proId);
42
43                  /* db.txt파일을 열고 통보문을 저장한후 파일닫기 */
44                  file = fopen("./db.txt", "a+");
45                  fprintf(file, "plusSem 프로쎄스통보문보관\n");
46                  fclose(file);
47
48                  /* 신호기의 값을 증가시킨 다음 1s 정지. */
49                  semB.sem_op = 1;
50                  if(semop(semId, &semB, 1) == -1)
51                  {
52                          fprintf(stderr, "신호기의 값증가에 실패\n");
53                          exit(0);
54                  }
55                  sleep(1);
56          }
57
58          /* 신호기를 제거 */
59          if(semctl(semId, 0, IPC_RMID, 0) == -1)
60          {
61                  fprintf(stderr, "신호기의 제거에 실패\n");
62                  exit(0);
63          }
```

64	
65	exit(1);
66	}

이번에는 minusSem프로그람의 원천코드를 보도록 하자.

실례프로그람: minusSem.c

```
1   #include <stdio.h>
2   #include <unistd.h>
3   #include <stdlib.h>
4
5   /* 신호기를 위한 머리부파일들 */
6   #include <sys/types.h>
7   #include <sys/ipc.h>
8   #include <sys/sem.h>
9
10  /* 신호기의 값을 감소시키는 main함수 */
11  int main(int argc, char *argv[] )
12  {
13          /* 신호기와 파일을 위한 변수선언 */
14          int step;
15          int semId, proId;
16          FILE *file;
17          struct sembuf semB;
18
19          /* sembuf의 초기값을 설정 */
20          semB.sem_flg = SEM_UNDO;
21          semB.sem_num = 0;
22
23          /* semget()를 리용하여 신호기의 ID를 얻기 */
24          semId = semget((key_t)1234, 1, 0666 | IPC_CREAT);
25
26          /* 신호기의 초기값을 설정 */
27          if(semctl(semId, 0, SETVAL, 1) == -1)
28          {
29                  fprintf(stderr, "신호기의 초기화에 실패\n");
30                  exit(0);
31          }
32
33          /* minusSem프로쎄스의 PID를 출력 */
```

```
34         printf("MINUS 신호기의 PID: %d\n", getpid());
35
36         /* 파일에 통보문을 5번 기입한후 완료 */
37         for(step = 0; step < 5; step++)
38         {
39                 /* 신호기에 마지막으로 수정을 가한 프로세스의 PID를 출력 */
40                 proId = semctl(semId, 0, GETPID, 0);
41                 printf("신호기를 변경한 마지막 PID: %d\n",proId);
42
43                 /* 신호기의 값을 감소시킨다. */
44                 semB.sem_op = -1;
45                 if(semop(semId, &semB, 1) == -1)
46                 {
47                         fprintf(stderr, "신호기값의 감소에 실패\n");
48                         exit(0);
49                 }
50
51                 /* db.txt파일을 열고 통보문을 보관한후 파일닫기 */
52                 file = fopen("./db.txt", "a+");
53                 fprintf(file, "minusSem프로세스 통보문 저장\n");
54                 fclose(file);
55
56                 sleep(1);
57         }
58         exit(1);
59 }
```

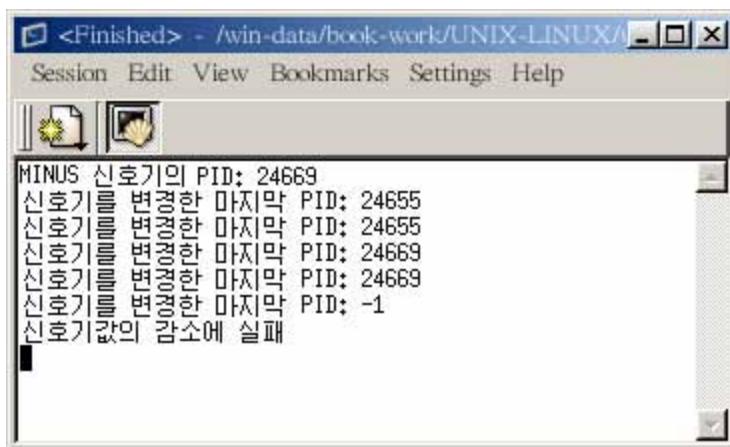



그림 5-8. minusSem.c 의 실행결과

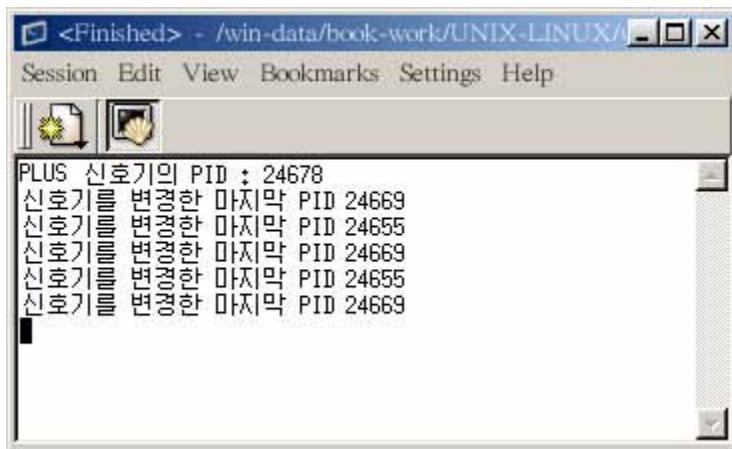



그림 5-9. plusSem.c 의 실행결과

원천코드를 모두 작성했으면 두개의 쉘창에서 개개의 프로그람을 실행시켜보자. 이때 시험을 위해 minusSem프로그람으로부터 실행시켜보자. plusSem프로그람의 경우에는 다른 프로쎄스의 도움이 없이도 작업수행이 가능하기때문에 plusSem프로그람을 먼저 실행시키면 혼자 작업을 마치고 탈퇴하게 된다.

프로그람을 실행시키면 그림과 같이(그림 5-8,9) 상대방의 프로쎄스에서 변경한 신호기의 내용이 자신에게 적용되였다는것을 확인할수 있다.

5.4.4. 신호기의 제거

UNIX는 현재 사용하고있는 IPC방법을 확인하거나 사용하지 않는 IPC방법을 제거할수 있는 쉘지령어를 제공하고있다. 현재 등록된 IPC방법들을 확인하는 지령은 ipcs이다.

그림 5-10에 ipcs지령을 실행시킨 실례를 보여주고있다.

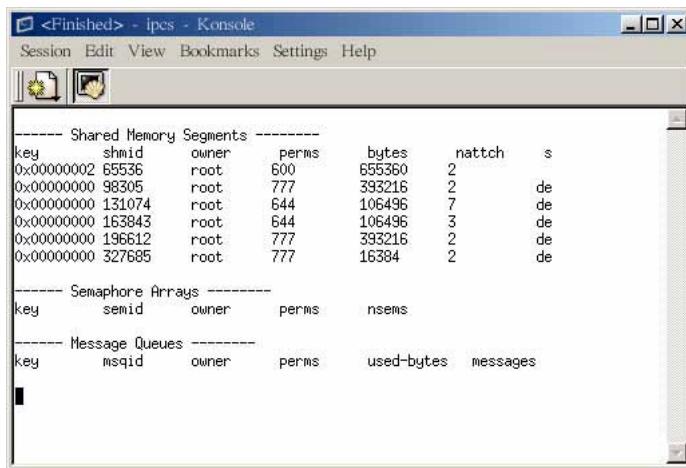



그림 5-10. ipcs 의 실행결과

여기서 실행결과를 자세히 보면 신호기의 열쇠값(key)이 나오는것을 볼수 있다. 이값은 16진수로서 리용하려고 할 때에는 10진수로 바꾸어 리용하면 된다.

만일 3456을 열쇠값으로 한 신호기를 쉘상에서 강제로 제거하려면 ipcrm지령을 리용하여 다음과 같이 실행하면 된다.

```
% ipcrm -S 3456
```

여기서 -S는 추가선택항목으로서 열쇠값을 리용하여 신호기를 삭제하는것을 의미한다. 만일 신호기의 ID를 리용하려면 -s(소문자)추가선택항목을 리용하면 된다.

의뢰기-봉사기모형

일반적으로 정보나 자원을 일률적으로 관리하고 제공하는 역할을 하는 콤퓨터(하드웨어나 쏘프트웨어)를 봉사기라고 하며 봉사기에 정보나 자원을 요청하고 그를 리용하는 역할을 하는 콤퓨터(하드웨어나 쏘프트웨어)를 의뢰기라고 하는데 이것들이 서로 련관되여 하나의 응용프로그람을 효률적으로 실행하도록 설계된 모형을 말한다.

이것들은 통신규약에 따라 통신을 진행하며 의뢰기측의 응용은 봉사기정보를 수시로 입수하면서 진행되게 된다.

제5절. 레코드잠그기

프로쎄스에서 동기화문제를 해결하기 위해 리용되는 다른 방법으로는 레코드잠그기(Record Locking)가 있다. 레코드잠그기는 프로쎄스들사이의 통신을 리용한 방법이 아니라 프로쎄스가 사용하려고 하는 자원을 잠시 보존할수 있도록 하는 방법이다. 이것을 리용하면 프로쎄스들사이에서 발생할수 있는 자원공유문제를 어느정도 해결할수 있다.

례를 들어 한 프로쎄스가 자료를 수정보충하거나 자료를 옮기는 과정에 다른 프로쎄스가 자료를 갱신하거나 삭제 및 변경을 한다면 이 프로쎄스는 전혀 다른 자료를 가져가거나 깨진 자료를 리용하게 된다. 이런 문제가 제기되면 발생원인을 찾기가 쉽지 않다. 프로쎄스를 개발한 사람들이 프로그람의 원천코드를 아무리 검사하여도 코드에는 문제가 없기때문이다. 이것은 여러개의 프로쎄스가 동시에 기동되면서 발생한 문제이므로 개발자들이 모여 동일한 환경을 만들고 오유를 발생시켜야 비로소 오유원인을 찾을수 있다. 따라서 이런 오유의 해결방도는 미리 예방하는것이 최선의 방도이다.

레코드잠그기는 다른 프로쎄스들에 현재 자원이 사용중임을 알리는 방법으로서 문제가 생기지 않도록 미리 예방하도록 해준다. 이때 레코드잠그기에서의 레코드는 파일이나 자원의 일부를 의미하며 잠그기는 다른 프로쎄스가 접근하지 못하도록 막는다는것을 의미한다. 레코드잠그기는 여러가지 방법으로 실현할수 있다.

5.5.1. lockf

lockf()체계호출함수를 리용하여 레코드잠그기를 실현할수 있다. 이 체계호출함수는 레코드잠그기를 아주 쉽게 실현할수 있도록 하는것이다. lockf()의 사용형식은 다음과 같다.

```
int lockf(int fileDes, int purpose, int size);
```

여기서 첫번째 인수인 fileDes는 lockf()호출이전에 열기한 파일서술자를 의미한다. 그리고 두번째 인수인 purpose는 lockf의 수행목적을 지정하는 인수이다. 마지막 인수는 sizs는 레코드잠그기에서 사용할 령역에 대한 크기를 의미한다.

두번째 인수인 purpose를 리용하여 lockf()의 수행목적을 지정한다고 했는데 이때 자주 사용되는 선언자는 다음과 같다.

- F_ULOCK(0): 잠그기의 해제
- F_LOCK(1): 잠그기의 수행

레코드잠그기의 크기를 지정하는 마지막 인수의 경우 시작위치는 현재 파일에 대한 지적자의 위치가 되기때문에 lseek()등으로 지적자의 위치가 옮겨가게 되면 거기서부터 지정한 크기까지가 레코드잠그기의 범위로 된다. 례를 들어 다음과 같이 레코드령역(1024Bytes)에 대해 잠그기를 진행할수 있다.

```
lockf(fileDes, F_LOCK, 1024L);
```

만일 크기를 0으로 설정하게 되면 현재의 파일지적자에서 파일의 끝까지가 령역으로 설정되는데 만일 현재의 지적자가 파일의 시작부분이라면 파일 전체가 해당 령역으로 설정된다. 여러개의 프로쎄스가 동일한 레코드령역을 리용해야 한다면 LOCK와 ULOCK을 적절히 리용하여 자원의 공유로 인해 발생할수 있는 문제를 예방할수 있다.

5.5.2. fcntl

이번에는 fcntl()체계호출함수를 리용하여 레코드잠그기를 실현하는 방법에 대해 보도록 하자. 앞에서 관을 리용하면서 fcntl()을 소개한바가 있다. 이것은 관의 블로크를 해제하는 방법을 소개하면서 나왔었다. 그때 소개된 문법은 다음과 같다.

```
#include <fcntl.h>
fcntl(fileDes, F_SETFL, O_NDELAY);
```

fcntl()함수를 리용하면 레코드잠그기를 읽기에 대한것과 쓰기에 대한것으로 나누어 잠그기를 진행할수 있다. 그러면 먼저 레코드잠그기를 위해 사용되는 fcntl()의 사용형식을 보도록 하자.

```
int fcntl(int fileDes, int cmd, struct flock* lockT);
```

첫번째 인수인 fileDes는 미리 열려진 파일의 서술자가 된다. fcntl은 읽기잠그기와 쓰기잠그기를 지원한다고 했는데 읽기를 위한 잠그기를 위해서는 fleDes가 읽기전용 또는 읽기, 쓰기용으로 열려야 하고 잠그기를 하려면 fileDes가 쓰기전용 또는 읽기, 쓰기용으로 열려져야 한다.

두번째 인수인 cmd는 fcntl()이 어떤 작업을 수행하는가에 대하여 지정한다. fcntl()가 레코드잠그기를 수행하기 위해 사용되는 cmd값에는 다음과 같은것들이 있다.

- F_GETLK: 레코드잠그기의 정보를 얻어 온다.
- F_SETLK: 잠그기하거나 잠그기를 해제하는데 사용된다.
- F_SETLKW: 파일에 대한 잠그기를 진행한다. 이미 잠그기가 되여있으면 작업을 중지한다.

세번째 인수인 lockT는 잠그기에 대한 정보를 얻어 오거나 잠그기할 때 개개의 정보를 담는 구조체로 활용된다. lockT의 자료형인 flock은 다음과 같이 이루어져있다.

```
struct flock {
      off_t l_start ;
      off_t l_Ien ;
      pid_t l_pid;
      short l_type;
      short l_whence;
   };
```

구조체속에 있는 l_start, l_whence 그리고 l_len는 레코드잠그기의 령역에 대한 정보를 얻는데 사용된다. 이때 l_whence와 l_start를 리용하여 파일의 정확한 시작위치를 구할수 있는데 l_whence는 l_start가 가리키는 지점이 파일지적자의 위치(처음인지 중간인지 끝인지)를 알려주고 l_len은 구역의 크기로 된다. 만일 l_len이 0이면 파일의 끝까지를 의미한다.

그리고 l_pid는 잠그기한 프로쎄스의 PID인데 이 값은 F_GETLK을 리용하여 레코드잠그기의 정보를 얻어올 때 의미가 있다. 마지막으로 l_type는 수행하려고 하는 잠그기의 형을 지정한다. 이때 사용되는 선언에는 다음과 같은것들이 있다.

- F_RDLCK: 읽기잠그기를 의미한다.
- F_WRLCK: 쓰기잠그기를 의미한다.
- F_UNLCK: 잠그기의 해제를 의미한다.

알아두어야 할것은 읽기잠그기를 하면 다른 프로쎄스는 쓰기잠그기를 못하게 되며 쓰기잠그기를 하면 읽기, 쓰기 잠그기를 모두 못하게 된다는것이다.

5.5.3. 레코드잠그기의 실례

이번에는 lockf()체계호출함수를 리용하여 레코드잠그기를 실현한 실례프로그람을 만들어보자. 여기에 나오는 례제는 신호기에서 나왔던 실례와 류사하다. 즉 db.txt파일을 열어 통보문을 입력하는 프로그람으로 통보문입력을 한후에 lock와 unlock를 수행하게 된다.

알아두어야 할것은 신호기를 위한 실례에서는 표준입출력서고인 fopen()을 리용했지만 lockf()체계호출함수에서는 open()함수를 리용하여 파일의 열기를 해야 한다는것이다.

그럼 전체 원천코드를 보도록 하자.

실례프로그람: recLock.c

```
1   #include <stdio.h>
2   #include <unistd.h>
3   #include <stdlib.h>
4   #include <fcntl.h>
5
6   int main(int argc, char *argv[])
7   {
8           /* 파일을 위한 변수선언 */
9           int file, step;
10
11          /* 파일에 통보문을 5번 기입한후 완료 */
12          for(step = 0; step < 5; step++)
13          {
14                  /* db.txt파일을 열기 */
15                  file = open("./db.txt", O_WRONLY | O_APPEND);
16                  if(file == -1)
17                  {
18                          fprintf(stderr, "파일의 열기에 실패\n");
19                          exit(0);
20                  }
21
22                  /* lockf()를 리용하여 lock를 수행한후 통보문을 입력 */
23                  lockf(file, F_LOCK, 0L);
24
25                  /* lockf()를 리용하여 unlock시킨후 파일닫기 */
26                  lockf(file, F_ULOCK, 0L);
27                  close(file);
28                  sleep(1);
29          }
30          exit(1);
31  }
```

제6장

프로세스들사이 통신2 (공유기억기와 통보대기렬)

서론

이 장에서는 5장에 이어 프로세스들사이의 통신에 대하여 계속해서 보기로 한다. 여기에서 취급할 주요내용은 공유기억기와 통보대기렬에 대한것으로서 모두 IPC를 실현하기 위해 많이 사용되는 중요한 방법들이다.

공유기억기는 토막(segment)이라고 부르는 특정한 기억기령역을 여러 프로세스가 함께 사용하는것을 의미하는데 이를 통하여 여러 프로세스들이 통신을 진행한다. 즉 기억기의 한 부분을 한 프로세스가 입력하면 다른 프로세스가 그 부분을 가져가는 방법을 리용하는것이다. 이때 프로세스들은 자기가 가지고있는 기억기에서 자료를 읽고쓰는것으로 인식하므로 처리속도가 매우 빠르다.

통보대기렬은 프로세스가 대기렬속에 통보문을 넣어주면 다른 프로세스가 대기렬속에 있는 통보문을 가져가는 방법으로 프로세스들 사이의 통신을 실현하는 방법이다. 통보대기렬은 Linux체계개발자들이 자주 리용하는 방법들중의 하나이다.

목표

1. 공유기억기
2. 통보대기렬
3. C++ 언어를 리용하여 IPC를 실현하는 프로그람작성

제1절. 공유기억기

6.1.1. 공유기억기란 무엇인가?

공유기억기(Shared Memory)는 이름 그대로 프로세스들이 특정한 기억기령역을 공유하도록 만든 다음 이 공간을 리용하여 통신을 진행할수 있도록 하는 기억기령역을 말한다. 기억기를 서로 공유하는 프로세스들은 공유가상기억기를 가리키는 표항목을 가지게 된다.

다른 IPC방법들과 마찬가지로 공유기억기는 열쇠값을 리용하여 접근하고 관리한다. 한편 공유기억기도 프로세스의 동기화가 필요하기때문에 신호기를 리용하여 자원에 대한 관리를 해주어야 한다.

Linux체계에서는 shm_segs라는 벡토르를 리용하여 공유기억기를 관리하게 된다. 그리고 벡토르속에는 shmid_ds라는 구조체(struct)가 보관되는데 shmid_ds를 리용하여 공유기억기정보를 보관하게 된다.

Linux 체계가 제공하는 체계호출함수를 리용하여 프로세스들은 공유기억기로 설정된 가상기억기를 사용할수 있게 된다. 이때 공유기억기와 련결된 가상기억기는 프로세스가 가진 가상기억기공간에 위치할수도 있고 Linux 체계가 가지는 별도의 령역에 위치할수도 있다.

프로세스가 공유기억기를 사용하려고 하면 핵심부는 해당 공유기억기에 대한 shmid_ds 구조체의 표항목을 찾아서 공유하고있는 가상기억기에 대한 정보를 얻게 된다. 만일 해당 령역이 존재하지 않으면 새로운 물리적기억기를 할당받아 표항목을 만들고 이것을 shmid_ds 구조체(struct)내부에 보관한다.

새로운 기억기령역을 할당받는 작업은 공유기억기를 사용하려고 하는 첫번째 프로세스에 의해 이루어지게 되며 두번째 프로세스부터는 단지 이 령역에 대한 정보를 자신의 가상기억기령역에 추가하여 사용하면 된다. 프로세스들이 공유기억기의 사용을 더 이상 원하지 않으면 공유기억기와 가상기억기의 련결은 끊어지게 된다.

프로세스와 공유기억기의 련결이 끊어지게 되면 shmid_ds자료구조체에 해당한 정보가 갱신된다. 이때 공유기억기를 사용하고있는 다른 프로세스는 특별한 영향을 받지 않는다. 하지만 모든 프로세스가 공유기억기와의 련결을 해제하게 되면 공유기억기를 위해 사용되던 기억기페지는 모두 해제되고 해당 shmid_ds구조체(struct)의 내용도 없어지게 된다.

여기서 알아두어야 할것은 공유기억기는 기억기령역에 대한 동기화를 제공하지 않으므로 개발자들은 이 부분에 신경을 써야 한다는것이다.

6.1.2. 공유기억기의 체계호출함수

공유기억기를 사용하기 위하여 다음과 같은 체계호출함수가 제공되고있다.

```
#include <sys/shm.h>
shmget(), *shmat(), shmdt(), shmctl()
```

shmget

shmget()함수는 공유기억기를 생성하고 공유기억기서술자를 얻는데 사용된다. 만일 동일한 공유기억기가 이미 존재하면 공유기억기를 새로 만드는 작업은 하지않고 이미 있던 공유기억기를 사용하게 된다. Shruget()함수의 사용형식은 다음과 같다.

```
int shmget(key_t key, size_t size, int shmflg);
```

여기서 shmget()함수가 사용하는 첫번째 인수인 key는 여러개의 프로쎄스들이 공통의 공유기억기를 사용할수 있도록 지정하는 열쇠값이다. 그리고 두번째 인수인 size는 공유기억기의 토막크기를 나타낸다. 마지막 인수인 shmflg는 공유기억기의 접근권한을 위한 기발로서 다른 기발들과 '|'기호를 리용하여 론리합연산을 수행할수 있다. 만일 IPC_CREAT기발과 함께 설정하게 되면 공유기억기가 생성되여 있지 않을 때 새로 생성되도록 할수 있다.

이때 만일 이미 생성된 공유기억기가 있다면 IPC_CREAT에 대한 설정은 무시되고 이미 만들어졌던 공유기억기가 반환된다. shmget() 함수는 실행에 실패하게 되면 -1을 되돌려주게 된다.

shmget()함수의 호출로 생성되는 공유기억기의 기억기령역은 프로쎄스의 가상기억령역이 아닌 실제기억기의 물리적령역이 된다. 그러면 shmget()체계호출함수의 간단한 사용실례를 보도록 하자.

원천코드:	
1	#include <sys/types.h>
2	#include <sys/ipc.h>
3	#include <sys/shm.h>
4	
5	/* 공유기억기로 사용할 크기를 선언 */
6	#define COMMANDSIZ 64
7	
8	/* shmget을 리용하여 공유기억기 확보 */
9	int smId = shmget((key_t)9000, COMMANDSIZ, 0666 | IPC_CREAT);
10	if (smId == -1)
11	{
12	printf("shmget실행이 실패\n");
13	exit(0);
14	}

shmat

shmget()체계호출함수를 리용하여 물리적기억기에 공유기억기를 생성했거나 이전에 생성되여있던 공유기억기정보를 얻어 왔다면 이번에는 shmat()체계호출함수를 리용하여 프로쎄스안에 있는 가상기억기와 련결시켜주어야 한다. shmat()함수의 사용형식은 다음과 같다.

```
void *shmat(int smId, const void *shm_addr, int flag);
```

shmat()함수를 리용하여 가상기억기와 물리적령역을 련결한 다음에야 공유기억기를 활용할수 있다. 이 shmat()함수의 첫번째 인수인 smId는 shmget()함수호출을 통해 얻어온 공유기억기의 서술자이다.

두번째 인수인 shm_addr은 공유기억기와 련결하려고 하는 프로쎄스안의 기억기를 가리킨다. 하지만 프로쎄스안의 기억기령역을 개발자가 지정해주는것보다 체계가 정황에 맞게 지정하도록 만들어주는것이 좋기때문에 일반적으로 0으로 설정한다.

마지막 인수로 사용하는 flag는 공유기억기와 프로쎄스안의 기억기를 련결하여 요구하는 속성을 지정한 기발로서 론리합연산을 통한 비트설정을 할수 있다. 이때 사용할수 있는 선언자에는 공유기억기와 련결되는 프로쎄스의 내부주소를 체계가 관리하도록 만들어주는 SHM_RND와 읽기전용으로 기억기련결을 지정하는 SHM_RDONLY 그리고 련결령역을 대치하는 SHM_REMAP 등이 있다.

shmat()함수는 공유기억기와 련결이 완료된 프로쎄스안의 가상기억기주소를 되돌려 준다. 만일 실패하면 -1을 돌려주게 된다. 다음 shmat()함수를 리용하여 기억기주소를 얻어내고 이것을 리용하기 위해 프로그람안의 완충기와 련결하는 간단한 모듈을 보도록 하자.

실례프로그람:modeChange.c	
1	#include <sys/types.h>
2	#include <sys/ipc.h>
3	#include <sys/shm.h>
4	
5	/* 공유기억기로 사용할 크기를 선언 */
6	#define COMMANDSIZ 64
7	
8	/* 공유기억기사용을 위한 변수선언 */
9	void *s_memory = (void *)0;
10	
11	/* 먼저 shmget()함수를 리용하여 공유기억기의 ID를 구한다. */
12	int smId = shmget((key_t)9000, COMMANDSIZ, 0666 | IPC_CREAT);
13	
14	/* shmat을 리용하여 공유기억기의 주소얻기 */
15	s_memory = shmat(smId, (void *)0, 0);

16	if(s_memory == (void *)-1)
17	{
18	printf("shmat실행 실패\n");
19	exit(0);
20	}
21	
22	/* 공유기억기주소와 내부변수의 지적자련결 */
23	char *buffer = (char *)s_memory;

shmdt

프로쎄스가 작업을 끝내고 더이상 공유기억기를 사용하지 않을 때에는 공유기억기와 프로쎄스안의 가상기억기사이의 련결을 끊어야 한다.

즉 공유기억기를 위해 사용되던 가상기억기를 제거하는 작업이 필요하다. 이때 사용하는 체계호출함수가 바로 shmdt()인데 이 함수의 사용형식은 다음과 같다.

```
int shmdt(const void *shm_addr);
```

shmdt()함수가 실행하면 프로쎄스안의 가상기억기공간은 해제되며 0을 되돌려주게 된다. 만일 실행에 실패하면 -1을 되돌려주게 된다. 그러면 앞에서 실행했던 shmat()을 리용하여 련결된 기억기령역을 해제하는 원천코드를 보도록 하자.

원천코드:	
1	#include <sys/types.h>
2	#include <sys/ipc.h>
3	#include <sys/shm.h>
4	
5	int smId = shmget((key_t)9000, COMMANDSIZ, 0666 | IPC_CREAT);
6	void *s_memory = shmat(smId, (void *)0, 0);
7	
8	/* 프로쎄스와 공유기억기를 분리 */
9	if(shmdt(s_memory) == -1)
10	{
11	printf("shmdt실행 실패\n");
12	exit(0);
13	}

shmctl

공유기억기를 할당받고 이것을 프로쎄스안의 가상기억기와 련결하는 등의 과정을 통하여 공유기억기령역을 사용할수 있다. 하지만 공유기억기자체에 접근해서 요구하는 설정

을 한다든가, 공유기억기의 제거 등의 작업이 필요할수도 있다. 이때 사용하는 체계호출함수가 shmctl()이다.

Shared Memory Control이라는 이름을 가진 shmctl()함수를 리용하여 공유기억기를 직접 조종할수 있는데 shmctl()함수의 사용형식은 다음과 같다.

```
int Shmctl(int smId, int cmd, struct shmid_ds *buf);
```

Shmctl()함수의 첫번째 인수인 smId는 shmget()함수의 호출로서 얻은 공유기억기의 ID이다. 두번째 인수인 cmd는 수행하려고 하는 지령을 의미하는데 다음과 같은 선언자들이 사용될수 있다.

- IPC_RMID: 공유기억기의 삭제에 리용
- IPC_SET: 마지막 인수인 buf의 값을 리용하여 공유기억기의 값을 설정한다.
- IPC_STAT: 공유기억기에 설정된 값을 buf안에 입력한다.

마지막 인수인 buf는 앞에서 소개한것처럼 공유기억기에 대한 각종 정보가 입력되는 구조체이다.

```
struct shmid_ds {
        struct ipc_perm shm_perm;
        size_t shm_segsz;
        ... ... ...
        pid_t shm_lpid;
        pid_t shm_cpid;
        ... ... ...
        time_t shm_ctime;
};
```

다음은 공유기억기를 생성한 다음 shmctl()함수를 리용하여 삭제하는 모듈을 간단히 살펴보자.

원천코드:	
1	#include <sys/types.h>
2	#include <sys/ipc.h>
3	#include <sys/shm.h>
4	
5	/* shmget을 리용하여 공유기억기 확보 */
6	int smId = shmget((key_t)9000, COMMANDSIZ, 0666 | IPC_CREAT);
7	
8	/* 공유기억기 제거 */
9	if(shmctl(smId, IPC_RMID, 0) == -1)

10	{
11	printf("shmctl실행 실패\n");
12	exit(0);
13	}

모듈에서 본바와 같이 shmctl()함수는 실행에 실패하면 -1을 되돌려주게 된다. 공유기억기의 삭제는 쉘상에서도 할수 있다. 이것은 앞에서 설명한 신호기의 제거에서 소개되였던 방법과 같은 방법으로 사용한다.

즉 ipcs지령의 실행을 통해 체계상에 설정된 IPC장치들의 정보를 얻을수 있다. 그림 6-1은 ipcs지령의 실행에 대한 실례를 보여주고있다.

그림 6-1. ipcs 지령의 실행

이때 공유기억기에 대한 내용만 확인하고 싶다면 다음과 같이 지령을 주면 된다.

```
#ipcs -m
```

이때 공유기억기를 삭제하려면 ipcrm지령을 리용한다. ipcrm지령을 shm추가선택과 공유기억기 ID와 함께 사용해서 실행하면 된다. 실례를 들면 다음과 같다.

```
#ipcrm shm 0
```

6.1.3. 공유기억기를 리용한 프로그람작성실례

지금까지 설명한 체계호출함수들을 리용하여 공유기억기에 대한 실례프로그람을 작성해보자. 이 실례프로그람은 호상 독립적으로 동작하는 두개의 프로쎄스가 공유기억기를 리용하여 통보문을 주고받는 프로그람이다. 프로쎄스들이 공유기억기를 리용할 때 주의해야 할 부분은 현재 공유기억기속에 있는 자료를 사용해도 되는것인지 혹은 이전에 이미 사용했던 자료가 아닌가에 대한 확인이 필요하다는것이다.

이를 위해 대부분의 개발자들이 약속된 구조를 리용하여 공유기억기령역에 접근하는 방법을 사용하고있다. 즉 구조체(struct)안의 어느 한 부분에 기발을 사용하여 기발이 on이면 자료를 가져가고 off이면 자료를 가져가지 못하게 한다는것이다.

또한 공유기억기를 통해 여러개의 프로쎄스가 작업을 진행하는 경우 프로쎄스동기화가 제기된다. 따라서 IPC를 리용하여 체계를 개발하는 개발자들은 항상 이러한 문제에 대해 신경을 써야 한다.

다음 실례에서는 공유기억기를 사용하는 프로쎄스들이 완충기의 첫번째 2 Byte를 기발로 사용한다. 즉 통보문을 입력하는 프로쎄스는 공유기억기의 첫번째 2Byte가 'NO'일 때 지령을 실행하기 위한 통보문을 공유기억기속에 입력한 다음 기발을 'ON'으로 변경시킨다.

공유기억기에서 실행할 지령을 접수하는 프로쎄스는 기발이 'ON'일 때 지령을 접수한 다음 기발을 'NO'로 변경하고 접수된 지령을 실행한다. 통보문을 입력하는 프로쎄스는 기발이 'NO'이면 다시 통보문을 입력하게 된다. 이러한 과정을 서로 반복하면서 언제 새로운 통보문을 접수해야 하는지 그리고 언제 통보문을 입력해야 하는가에 대해 알게 된다.

지령을 입력받는 rcvSm프로그람은 공유기억기의 생성과 완료를 담당하게 된다. 따라서 먼저 실행되여 지령이 입력되기를 기다리고 있어야 한다. 통보문을 전송하는 sendSm프로그람은 사용자로부터 문장을 입력받은 다음 해당 문장을 공유기억기에 기입한다. 만일 사용자가 'quit'을 입력하면 sendSm프로쎄스와 rcvSm프로쎄스는 차례로 완료하게 된다.

그러면 먼저 rcvSm프로그람의 작성과정을 살펴보도록 하자. 이 프로그람은 shmget()함수를 리용하여 공유기억기를 만든다. 그 다음 shmat()함수를 리용하여 공유기억기와 프로쎄스안의 기억기에 련결을 설정한다.

원천코드:	
1	#define COMMANDSIZ 64
2	/* shmget을 리용하여 공유기억기 확보 */
3	int smId = shmget((key_t)9000, COMMANDSIZ, 0666 | IPC_CREAT);
4	/* shmgat을 리용하여 공유기억기주소 얻기 */
5	void *s_memory = shmat(smId, (void *)0, 0);

그 다음 완충기를 리용하여 프로쎄스안의 기억기를 조작할수 있도록 만든다. 모든 준비작업이 끝나면 완충기의 첫번째 2Byte가 ON이 되기를 기다린다. 만일 ON이면 통보문을 접수하고 완충기의 첫번째 2Byte를 NO로 설정한다.

원천코드:	
1	/* 공유기억기주소와 내부변수의 지적자련결 */
2	char *buffer = (char *)s_memory;
3	/* 기발이 ON이면 통보문을 접수 */
4	if(!strncmp(buffer, "ON", 2))
5	{
6	printf("지령접수: %s\n", buffer+2);
7	/* 기발이 ON에서 NO로 변경 */
8	strncpy(buffer, "NO", 2);
9	}

만일 접수된 통보문이 'quit'이면 프로쎄스와 공유기억기를 분리시키는 작업을 수행한다. 그 다음 공유기억기를 체계에서 제거한다.

원천코드:	
1	/* 프로쎄스와 공유기억기를 분리 */
2	shmdt(s_memory);
3	/* 공유기억기 제거 */
4	shmctl(smId, IPC_RMID, 0);

이번에는 공유기억기속에 통보문을 입력하는 sendSm프로그람에 대해 보도록 하자. sendSm프로그람의 첫번째 작업도 shmget()을 리용하여 공유기억기의 서술자를 가져온다. 이때 얻어온 서술자를 리용하여 프로쎄스안의 기억기와 공유기억기를 련결시킨다. 그 다음 사용할 완충기와 기억기를 련결시키고 통보문을 입력하게 된다.

이때 완충기의 기발이 ON이 아니면 통보문을 입력시키고 ON이면 rcvSm프로쎄스가 ON을 해제할 때까지 기다리게 된다. 만일 사용자가 입력한 통보문이 'quit'이면 공유기억기의 사용을 해제한 후 프로쎄스를 완료하게 된다.

원천코드:	
1	/* 통보문으로 주고받을 자료의 길이와 형 정의 */
2	#define BUFLEN 32
3	typedef struct
4	{
5	long int msgType; /* 통보문 형 */
6	char userNo[13+1]; /* 사용자의 시민증번호 */
7	char address[17+1]; /* 사용자의 주소 */
8	} UserType;
9	
10	/* msgQID 얻기 */
11	msgQid = msgget ((key_t) 8000, 0666 | IPC_CREAT);
12	
13	/* 통보문 작성 */
14	UserType userT;
15	userT.msgType = 1;
16	strncpy(userT.userNo, "시민증번호...\0",13);
17	strncpy(userT.address, "주소...\0",17);
18	
19	/* 작성된 통보문 전송 */
20	result = msgsnd(msgQid, (void *)&userT,BUFLEN,0);
21	if(result == -1)
22	{
23	printf("msgsnd의 실행에 실패\n");
24	exit(0);
25	}

그러면 지금까지 설명한 내용에 기초하여 원천코드를 작성하여보자. 원천코드를 볼 때에는 분석과정을 구체적으로 따져보아야 한다. 코드안에 어떤 오유가 있는지, 어떤 점을 개선해야 하는지, 또한 어떤 부분을 앞으로 참고하겠는지 등을 생각하면서 보아야 앞으로 실지 프로그람작성에서 참고할수 있다. 먼저 rcvSm.c의 원천코드를 분석해보자.

실례프로그람: rcvSm.c	
1	#include <stdio.h>
2	#include <string.h>

```
3
4   #include <sys/types.h>
5   #include <sys/ipc.h>
6   #include <sys/shm.h>
7
8   /* 공유기억기로 사용할 크기를 선언 */
9   #define COMMANDSIZ 64
10
11  int main()
12  {
13          /* 공유기억기사용을 위한 변수선언 */
14          void *s_memory = (void *)0;
15          int smId;
16          char *buffer;
17          int isRun = 1;
18
19          /* shmget을 리용하여 공유기억기 확보 */
20          smId = shmget((key_t)9000, COMMANDSIZ, 0666 | IPC_CREAT);
21          if (smId == -1)
22          {
23                  printf("shmget실행 실패\n");
24                  return 0;
25          }
26
27          /* shmgat을 리용하여 공유기억기의 주소를 얻기 */
28          s_memory = shmat(smId, (void *)0, 0);
29          if(s_memory == (void *)-1)
30          {
31                  printf("shmat실행 실패\n");
32                  return 0;
33          }
34
35          /* 공유기억기주소와 내부변수의 지적자를 련결 */
36          buffer = (char *)s_memory;
37          while(isRun)
38          {
39                  /* ON이면 상대방이 넣어준 지령을 접수 */
```

40	if(!strncmp(buffer, "ON", 2))
41	{
42	printf("지령접수: %s\n", buffer+2);
43	/* 기발을 ON에서 NO로 변경 */
44	strncpy(buffer, "NO", 2);
45	/* 기발을 제외한 문자렬이 quit로 시작되면 완료 */
46	if(!strncmp(buffer+2, "quit", 4))
47	{
48	isRun = 0;
49	}
50	}
51	}
52	
53	/* 프로쎄스와 공유기억기를 분리 */
54	if(shmdt(s_memory) == -1)
55	{
56	printf("shmdt실행 실패\n");
57	return 0;
58	}
59	
60	/* 공유기억기 제거 */
61	if(shmctl(smId, IPC_RMID, 0) == -1)
62	{
63	printf("shmctl실행 실패\n");
64	return 0;
65	}
66	return 1;
67	}

이번에는 sendSm.c프로그람의 원천코드를 보도록 하자.

실례프로그람: sendSm.c	
1	#include <stdio.h>
2	#include <string.h>
3	
4	#include <sys/types.h>
5	#include <sys/ipc.h>

```
6	#include <sys/shm.h>
7	
8	/* 공유기억기로 사용할 크기를 선언 */
9	#define COMMANDSIZ 64
10	
11	int main()
12	{
13		/* 공유기억기사용을 위한 변수선언 */
14		void *s_memory = (void *)0;
15		int smId;
16		char *buffer;
17		int isRun = 1;
18	
19		/* shmget을 리용하여 공유기억기 확보 */
20		smId = shmget((key_t)9000, COMMANDSIZ, 0666 | IPC_CREAT);
21		if (smId == -1)
22		{
23			printf("shmget실행 실패\n");
24			return 0;
25		}
26	
27		/* shmgat을 리용하여 공유기억기의 주소를 얻기 */
28		s_memory = shmat(smId, (void *)0, 0);
29		if(s_memory == (void *)-1)
30		{
31			printf("shmat실행 실패\n");
32			return 0;
33		}
34	
35		/* 공유기억기주소와 내부변수의 지적자를 련결 */
36		buffer = (char *)s_memory;
37	
38		while(isRun)
39		{
40			/* ON이면 상대방이 가져갈 때까지 대기 */
41			while (strncmp(buffer, "ON", 2) == 0) {}
42			/* 공유기억기안에 지령행입력 */
```

43	printf("지령입력: ");
44	gets(buffer+2);
45	strncpy(buffer, "ON", 2);
46	/* ON문자렬뒤에 quit가 입력되여있으면 완료 */
47	if(!strncmp(buffer+2, "quit", 4))
48	{
49	isRun = 0;
50	}
51	}
52	
53	/* 프로쎄스와 공유기억기를 분리 */
54	if(shmdt(s_memory) == -1)
55	{
56	printf("shmdt실행 실패\n");
57	return 0;
58	}
59	
60	return 1;
61	}

프로그람작성이 모두 끝났으면 콤파일을 한 다음 두개의 쉘창에서 각각 실행을 시키도록 한다. 그림 6-2, 6-3은 매개 프로그람의 실행결과를 보여주고있다.

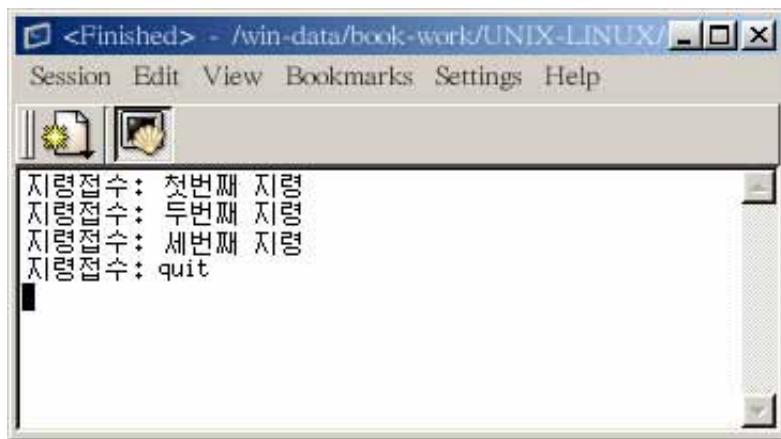



그림 6-2. rcvSm.c 의 실행결과

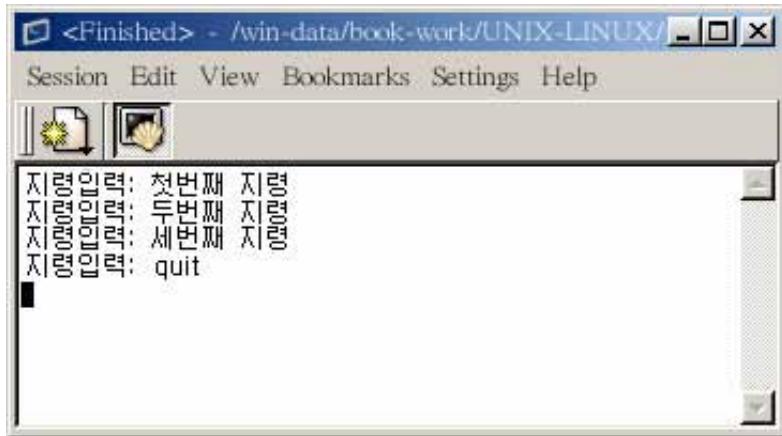



그림 6-3. sendSm.c 의 실행결과

상식

XML 이란 무엇인가?

XML(Extensible Markup Language)은 홈페지의 서술언어인 HTML(하이퍼본문표식언어)의 후계언어로서 SGML(표준일반화표식언어)의 확장기능을 웨브우에서도 리용할수 있게 한 언어이다.

1998 년 2 월에 W3C(WWW 협회)가 기본 기술적특성을 제정하였다. HTML 과의 기본 차이는 사용자가 독자적인 꼬리표를 사용하여 자료의 속성정보나 론리구조를 정의할수 있다는것이다. 또한 자료의 속성과 내용을 관련시켜 서술할수 있다. 례를 들면 《생년월일》이라는 꼬리표를 정의하고 그 꼬리표안에 《1999.01.01》이라는 자료를 포함시키면 1999.01.01 이라는 자료가 생년월일을 의미하게 된다.

XML(확장표식언어)로 서술된 자료는 업무체계에 직접 삽입되여 처리될수 있기때문에 기업들이 인터네트를 통하여 주문자료를 주고 받는 웨브전자자료교환에서의 자료형식으로서 리용되고있다.

제2절. 통보대기렬

이 절에서는 IPC의 방법들중에서 가장 많이 쓰는 통보대기렬에 대하여 간단히 보고 그것이 어떻게 사용되는가를 체계호출함수와 실례를 통하여 설명한다.

6.2.1. 통보대기렬의 간단한 소개

통보대기렬을 사용하여 IPC를 실현한다는것은 프로세스들이 함께 사용하는 통보대기렬에 통보문을 입출력하면서 서로 요구하는 자료를 주고받는다는것을 의미한다. 이를 위하여 체계는 대기렬을 관리해야 하고 통보대기렬속에 여러개의 프로세스가 접근할수 있도록 해주어야 한다.

알아둡시다

통보대기렬은 일반적으로 FIFO(First In FIrst Out)에 따르는 자료구조를 의미한다. 하지만 여기서는 FIFO가 체계의 기억기령역을 표현한다. 통보대기렬과 대조되는 자료구조로서 탄창이 있는데 탄창은 LIFO(Last In First Out)를 따른다.

Linux는 대기렬들을 msgque목록을 리용하여 관리한다. 이 목록속에는 msqid_ds 구조체들이 보관된다. msqid_ds구조체에는 대기렬에 대한 정보가 보관되는데 프로쎄스가 대기렬을 생성하면 새로운 msqid_ds가 생성된다. 그리고 생성된 msqid_ds는 msgque 목록안에 삽입된다.

msqid_ds구조체에는 대기렬의 접근권한이나 대기렬의 생성, 수정시간 등의 정보가 보관된다. 그리고 구조체안에는 두개의 대기렬을 가지고있다. 이것은 통보대기렬에서 정보를 읽으려고 하는 프로쎄스를 위한것과 통보대기렬에 정보를 입력하려는 프로쎄스를 위한것들이다. msqid_ds구조체를 보면 다음과 같다.

```
struct msgid_ds {
    Struct ipc_Perm msg_perm; /* 대기렬에 대한 접근권한 */
    Struct msg *msg_first; /* 대기렬속의 첫번째 통보문*/
    Struct msg *msg_last; /* 대기렬속의 마지막 통보문*/
    time_t msg_stime; /* 마지막에 통보문이 전송된 시간 */
    time_t msg_rtime; /* 마지막으로 통보문을 읽은 시간 */
    ... ... ...
    struct wait_queue *wait; /* write 프로쎄스를 위한 대기렬*/
    struct wait_queue *rwait; /* read 프로쎄스를 위한 대기렬 */
    ushort msg_cbytes; /*대기렬속에 있는 현재 통보문의 byte 수*/
    ushort msg_qnum; /* 대기렬속에 있는 현재 통보문의 개수 */
    ... ... ...
    ushort msg_lspid; /* 마지막으로 통보문을 전송한 PID */
    ushort msg_lrpid; /* 마지막으로 통보문을 수신한 PID */
    ... ... ...
  };
```

통보대기렬에 있는 통보문에 대한 읽기 또는 쓰기를 시도하면 체계는 msqid_ds구조체의 첫번째 요소인 msg_perm을 리용하여 대기렬에 대한 접근권한을 검사하게 된다. 만일 허용된 사용자나 그룹이면 해당 통보문은 작업을 시도한 프로쎄스에 전달된다.

대기렬을 리용하여 프로쎄스들이 작업을 수행하는 과정에는 통보문의 개수나 길이 등의 제한으로 작업을 중지하고 대기해야 하는 경우가 발생할수 있다. 이때에는 msqid_ds 구조체속의 대기렬을 리용하게 된다. 통보대기렬은 이러한 방식으로 내부적으로 동기화

문제를 해결한다. 즉 대기렬에서 한 프로쎄스가 작업기다림상태이면 내부적인 일정관리기에 의해 다른 프로쎄스가 작업을 끝냈을 때 해당 통보대기렬에 대한 사용권을 넘겨받는다. 그리고 자기의 작업이 끝나면 다른 프로쎄스에 작업권을 넘긴다.

그러면 통보대기렬의 생성과 사용,그리고 삭제와 관련한 체계호출함수에 대해 보도록 하자.

6.2.2. 통보대기렬 체계호출함수

Linux체계는 다음과 같은 체계호출함수들을 제공하고있다.

```
  int msgget(key_t key, int msgflg);
  int msgsnd(int msqid, void *msg_ptr, size_t msg_sz, int msgflg);
  int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int
msgtype,int msgflg);
  int msgctl(int msqid, int command, struct msqid_ds *buf);
```

그러면 매개 체계호출함수들에 대해 차례로 보자.

msgget

통보대기렬도 공유기억기나 신호기처럼 열쇠값을 리용하여 IPC장치를 리용한다. 이때 열쇠는 Linux체계가 IPC장치를 가리키는데 사용된다. 프로그람안에서 열쇠값을 지정하기 위하여 types.h머리부파일에 있는 key_t형의 구조체를 리용하게 된다.

따라서 IPC장치를 리용하려면 열쇠를 리용하여 IPC를 생성하여야 한다. 통보대기렬에서 이러한 작업을 수행하는 체계호출함수가 바로 msgget()이다. msgget()체계호출함수를 리용하여 열쇠값에 해당한 통보대기렬를 생성하게 되는데 이 함수는 다음과 같은 인수들을 리용한다.

```
key_t key
int flag;
int msgQid = msgget(key, int) ;
```

첫번째 인수로 사용하는 key가 바로 IPC장치를 위해 Linux체계가 사용할 열쇠이다. 두번째 인수인 flag는 대기렬에 대한 접근권한을 설정하는 기발과 OR연산자를 리용하여 생성을 위한 기발 등과 결합할수 있다. 아래의 실례는 열쇠값 8000인 통보대기렬 생성모듈이다.

```
int msgQid = msgget( (key_t)8000, 0666 | IPC_CREAT);
```

msgget()실행이 성공적으로 진행되면 통보대기렬에 대한 ID를 되돌려 주게되는데 이 ID를 리용하여 통보문전송이나 통보대기렬 조종 등을 하게 된다. 만일 msgget()의 실행이 실패하면 -1을 되돌리게 된다.

msgget의 실행을 통하여 통보문대기렬을 생성했으면 통보문전송을 위한 연산을 수행할수 있게 되는데 이때 사용하는 체계호출함수는 msgsnd()와 msgrcv()이다.

msgsnd

통보대기렬안에 통보문을 입력(write)하기 위해 사용하는 체계호출함수가 msgsnd()이다. 대기렬속에 자료를 입력하려고 할 때 체계는 통보대기렬에 대한 쓰기권한이 있는지 조사하게 된다.

이때 사용되는 정보는 앞에서 소개한 msqid_ds구조체의 ipc_perm의 꼬리표(tag)안에 있는 정보이다. 이 안의 정보와 비교하여 쓰기권한에 다른 문제가 없으면 msgsnd()함수를 제대로 수행할수 있게 된다.

msgsnd()함수의 간단한 사용형식은 다음과 같다.

```
int msgQid; /* 통보대기렬의 ID */
void *msgPtr; /* 전송하려고 하는 통보문의 지적자형변수 */
size_t msgSize; /* 전송하려고 하는 통보문의 크기*/
int msgFlag; /* 통보문전송을 위한 조종기발*/
int result = msgsnd(msqid, msgPtr, msgSize, msgFlag);
```

msgsnd()체계호출함수에서 사용되는 첫번째 인수인 msgQid는 사용하려고 하는 통보대기렬의 ID이고 두번째 인수인 msgPtr는 전송하려고 하는 통보문의 지적자형변수가 된다. 그리고 세번째 인수인 msgSize는 전송하려고 하는 통보문의 크기이다.

이때 사용하는 통보문의 형은 개발자들이 설계한 통보문의 형을 리용하면 된다. 이때 통보대기렬에 들어오는 통보문의 종류를 구분하여 프로쎄스들이 사용할수 있게 하려면 다음과 같이 통보문형의 첫번째 마당을 long int로 지정하여 사용해야 한다.

```
struct new_msg_t {
      long int msgType;
      ... ... ...
};
```

여기서 long int형으로 지정된 msgType를 제외한 통보문들은 전송크기를 지정해야 한다. 이렇게 msgType와 크기를 지정한후 통보문을 전송하면 통보문을 받는 프로쎄스는 msgType를 리용하여 지정된 통보문을 읽어가게 된다. 만일 이러한 규칙을 따르지 않으면 입력된 통보문을 조건에는 관계없이 그냥 전송하고 그냥 읽게 된다.

마지막으로 사용된 인수인 msgFlag는 통보문전송에 사용되는 조종기발로서 다음과 같은 값이 들어올수 있다.

- IPC_NOWAIT: 작업을 대기해야 하는 경우가 발생하면 대기하지 않고 바로 되돌리기
- IPC_NOERROR: 지정된 크기보다 더 큰 통보문이 전송되여도 오유가 나오지 않도록 설정한다. 이때 크기를 초과한 통보문은 삭제된다. 만일 이러한 설정이 없으면 오유를 되돌리기 한다.

이러한 기발은 론리합연산자를 리용하여 함께 지정할수 있다. 그러면 msgsnd의 간단한 사용실례를 보도록 하자. 다음의 실례는 8000번을 열쇠로 하여 통보대기렬을 생성한 다음 사용자의 시민증번호와 주소를 통보대기렬에 입력하는것을 보여주고있다. 이때 통보문형은 1로 한다.

원천코드:

```
1  /* 통보문으로 주고 받을 자료의 길이와 형 정의 */
2  #define BUFLEN 32
3  typedef struct
4  {
5          long int msgType;  /* 통보문 형 */
6          char userNo[13+1]; /* 사용자의 시민증 번호 */
7          char address[17+1]; /* 사용자의 주소 */
8  } UserType;
9
10 /* msgQID 얻기 */
11 msgQid = msgget ((key_t) 8000, 0666 | IPC_CREAT);
12
13 /* 통보문 작성 */
14 UserType userT;
15 userT.msgType = 1;
16 strncpy(userT.userNo, "시민증번호...\0",13);
17 strncpy(userT.address, "주소...\0",17);
18
19 /* 작성된 통보문 전송 */
20 result = msgsnd(msgQid, (void *)&userT,BUFLEN,0);
21 if(result == -1)
22 {
23 printf("msgsnd의 실행에 실패\n");
24 exit(0);
25 }
```

만일 msgsnd()를 리용한 통보문전송에 실패하면 -1이 되돌려진다. 이번에는 통보대기렬에서 통보문을 읽어들이는 msgrcv체계호출함수에 대해 보도록 하자.

msgrcv

msgsnd체계호출함수를 리용하여 입력된 통보대기렬속의 통보문은 msgrcv()체계호출함수를 리용하여 가져올수 있다. 통보문을 가져 온 다음에는 대기렬속에서 해당 통보문이 삭제된다. msgrcv() 함수의 사용형식은 다음과 같다.

```
int msgQid; /* 통보대기렬의 ID */
void *msgPtr; /* 전송하려고 하는 통보문의 지적자형변수*/
size_t msgSize; /* 전송하려고 하는 통보문의 크기*/
long int msgType; /* 통보문의 형*/
int msgFlag; /* 통보문전송을 위한 조종기발*/
int result = msgrcv(msgQid, msgPtr, msgSize, msgType, msgFlag);
```

msgrcv()함수에서 사용하는 첫번째 인수인 msgQid는 msgget()를 통해 얻어 온 대기렬의 ID이고 두번째 인수인 msgPtr은 대기렬에서 읽어올 통보문을 보관하는데 사용된다. 세번째 인수인 msgSize는 대기렬에서 읽어 올 통보문의 크기이다. 이때 통보문의 크기에서 long int형으로 정의된 첫번째 끄리표정보의 크기는 뺀다. 네번째로 사용되는 msgType인수는 대기렬에서 읽어 올 통보문의 형을 지정하는데 사용된다. 즉 여기서 지정한 값에 해당되는 통보문을 대기렬에서 가져오게 된다. 이때 대기렬안에서 사용되는 값은 msgsnd()가 대기렬에 전송한 통보문의 첫번째 끄리표인 long int형의 값이 활용된다.

msgrcv()를 실행하면서 msgType을 0으로 설정하면 통보문형에 상관없이 대기렬속에 보관된 첫번째 통보문을 읽어온다.

마지막 인수인 msgFlag에는 msgsnd()함수에서와 같이 IPC_NOWAIT 또는 MSG_NOERROR등을 사용하면 된다. msgrcv()함수가 성공적으로 실행되면 읽어 온 통보문의 바이트수를 되돌리게 된다. 만일 실패하면 -1을 되돌리게 된다. 그러면 간단한 실례를 통해 msgrcv()함수의 실행과정을 보도록 하자.

다음은 사용자의 시민증번호와 주소정보를 통보대기렬을 통해 입력받는 과정을 보여주고있다.

원천코드:	
1	/* 통보문으로 주고 받을 자료의 길이와 형 정의 */
2	#define BUFLEN 32
3	typedef struct
4	{
5	long int msgType; /* 통보문 형 */
6	char userNo[13+1]; /* 사용자의 시민증번호 */
7	char address[17+1]; /* 사용자의 주소 */
8	} UserType;
9	
10	/* 통보대기렬의 ID얻기 */
11	int msgQid = msgget ((key_t) 8000, 0666 | IPC_CREAT);
12	
13	/* 통보문 읽기 */

14	UserType userT;
15	long int msgType = 0;
16	int result = msgrcv(msgQid, (void *)&userT, BUFLEN, msgType, 0);
17	if(result == -1)
18	{
19	printf("msgrcv실행실패에 대한 오유번호: %d\n",errno);
20	exit(0);
21	}

msgctl

공유기억기의 조종을 담당했던 shmctl과 같은 기능을 가진 통보대기렬의 체계호출함수에는 msgctl이 있다. msgctl()함수는 통보대기렬을 제거하거나 대기렬의 현재상태를 검색하거나 또는 대기렬의 설정을 변경할 때 사용된다. 먼저 msgctl()의 사용형식을 보면 다음과 같다.

```
int msgctl(int msgQid, int cmd, struct msqid_ds *buf);
```

첫번째 인수로 사용된 msgQid는 msgget()함수를 리용하여 얻어온 대기렬의 ID이고 두번째 인수인 cmd는 msgctl을 통해 수행하려고 하는 지령이다. 이때 사용할수 있는 지령에는 다음과 같은것들이 있다.

- IPC_RMID: 대기렬을 삭제하기 위해 사용된다.
- IPC_STAT: 통보대기렬의 상태값을 마지막 인수인 buf를 통해 얻어온다.
- IPC_SET: 마지막 인수인 buf의 값을 리용하여 통보대기렬의 상태를 설정한다.

마지막 인수로 사용되는 buf는 처음에 소개했던 msqid_ds 구조체로서 IPC_STAT 또는 IPC_SET지령을 위해 사용된다.

그러면 다음의 실례를 통하여 msgctl()의 사용을 보도록 하자. 아래의 원천코드는 8000을 열쇠값으로 리용하는 통보대기렬를 제거하는 과정을 보여준다.

원천코드:	
1	/* 통보대기렬 ID얻기 */
2	int msgQid = msgget ((key_t) 8000, 0666 | IPC_CREAT);
3	
4	/* 통보대기렬을 제거한다. */
5	int result = msgctl(msgQid, IPC_RMID, 0);
6	if(result == -1)
7	{
8	printf("msgctl실행 실패\n");
9	exit(0);
10	}

우의 실례에서 알수 있는바와 같이 msgctl()의 실행이 실패하면 -1을 되돌려주게 된다.

그러면 지금까지 설명한 체계호출함수들을 리용하여 실례프로그람을 작성해 보자.

한가지 알아두어야 할것은 통보대기렬을 체계에서 확인하거나 제거하기 위해 쉘지령을 리용할수 있다는것이다. 이때 사용되는 지령으로서는 ipcs와 ipcrm 지령이 있다. ipcs 지령은 Linux체계에 등록된 IPC장치를 화면에 현시한다. 그리고 ipcrm 지령은 등록된 IPC 장치를 제거한다. 아래에서는 ipc지령을 리용하여 통보대기렬의 정보를 확인하고 제거하는 간단한 과정을 보여주었다.

```
#ipcs    q            /*통보대기렬에 대한 정보를 출력*/
#ipcrm    q <id>   /*통보대기렬의 ID를 리용하여 대기렬을 제거*/
#ipcrm msg <id>   /*-q 추가선택항목대신 msg를 리용*/
```

6.2.3. 통보대기렬 실례프로그람

통보대기렬를 리용하여 두개의 프로쎄스가 사용자의 정보를 주고받는 프로그람을 작성하여보자. 여기서 한 프로쎄스는 통보대기렬에 있는 자료를 읽어들이는 역할을 담당하고 또 다른 프로쎄스는 대기렬에 자료를 입력하는 역할을 하여야 한다.

이러한 프로그람구조는 실지 프로그람작성에서 많이 제기된다. 즉 특정한 프로쎄스는 사용자로부터 자료를 받는 일을 하고 다른 프로쎄스는 자료처리를 담당하게 만든다. 그리고 이러한 프로쎄스들은 통보대기렬을 리용하여 서로의 작업을 원활하게 수행한다. 이러한 구조(대기렬을 사이에 두고 다중프로쎄스로 작업을 처리하는 구조)를 가지면 여러가지 우점이 있다.

가령 자료처리가 지연되여 사용자가 자료를 받지 못하는 경우를 방지할수 있다. 그리고 사용자로부터 무한정 입력을 기다릴 필요도 없다. 대기렬에 자료가 없으면 다른 작업을 수행하고 자료가 입력되면 그때 처리를 해주면 되기때문이다.

그러면 자료의 입력을 기다리는 rcvMsg프로그람을 작성해보자. 먼저 통보대기렬을 통해 입력받을 자료의 형을 지정한다. 이때 통보문을 전송할 프로쎄스와 약속된 형을 사용해야 한다. 만일 전송할 자료형의 종류가 여러가지라면 이에 맞는 통보문형을 지정해주어야 한다.

원천코드:	
1	/* 통보문으로 주고받을 자료의 길이와 형 정의 */
2	#define BUFLEN 32
3	typedef struct
4	{
5	long int msgType; /* 통보문의 형 */
6	char userNo[13+1]; /* 사용자의 시민증번호 */

7	char address[17+1]; /* 사용자의 주소 */
8	} UserType;

통보문의 정의가 끝났으면 통보대기렬을 생성하고 대기렬의 ID를 얻도록 한다. 그 다음 대기렬의 ID를 리용하여 통보문을 읽어들인다.

```
/* 통보문대기렬 ID 를 얻는다*/
int msgQid = msgget((key_t)8000, 0666 | IPC_CREAT) ;
/* 통보문의 읽기 */
long int msgType = 1;
UserType userT;
int result = msgrcv(msgQid, (void *) & userT, BUFLEN, msgType, 0);
```

만일 읽어온 사용자의 정보(시민증번호)가 quit문자렬이면 읽기를 중단하고 통보대기렬을 제거하도록 한다.

```
if( !strncmp(userT.userNo,"quit" ,4))
{
작업탈퇴;
}
/* 통보대기렬을 제거한다. */
int result = msgctl(msgQid, IPC_RMID, 0);
```

이번에는 통보문을 전송하는 프로그람인 sendMsg.c를 작성해보자.

sendMsg에서 사용하는 자료형이나 통보대기렬의 ID를 얻어오는 방법은 동일하다. 그리고 rcvMsg프로쎄스가 통보대기렬을 제거하기때문에 특별히 통보대기렬을 제거하는 작업을 수행할 필요는 없다. 다만 다음과 같이 통보문을 전송하는 루틴(routine)이 sendMsg 프로그람의 주요모듈로 된다.

```
/* 통보문작성 */
UserType userT;
memset(&userT, '\0', BUFLEN) ;
userT.msgType = 1;
printf("시민증번호: " ); gets(buffer); strncpy(userT.userNo, buffer,13);
printf("주소: "); gets(buffer); strncpy(userT.address, buffer,17);

/* 작성된 통보문전송 */
      int result = msgsnd(msgQid, (void *)&userT, BUFLEN, 0);
```

그러면 지금까지 설명한 내용에 기초하여 프로그람을 작성하여 보자.

먼저 rcvMsg.c파일을 보면 다음과 같다.

실례프로그람: rcvMsg.c	
1	#include <stdio.h>
2	#include <string.h>
3	#include <errno.h>
4	
5	#include <sys/types.h>
6	#include <sys/ipc.h>
7	#include <sys/msg.h>
8	
9	/* 통보문으로 주고받을 자료의 길이와 형정의 */
10	#define BUFLEN 32
11	typedef struct
12	{
13	long int msgType; /* 통보문의 형 */
14	char userNo[13+1]; /* 사용자의 시민증번호 */
15	char address[17+1]; /* 사용자의 주소 */
16	} UserType;
17	
18	int main()
19	{
20	/* 필요한 변수선언 */
21	int isRun = 1;
22	int msgQid, result;
23	UserType userT;
24	long int msgType = 1;
25	
26	/* 통보대기렬의 ID 얻어오기 */
27	msgQid = msgget ((key_t) 8000, 0666 | IPC_CREAT);
28	if(msgQid == -1)
29	{
30	printf("msgget실행 실패오유번호: %d\n", errno);
31	return 0;

32	}
33	
34	/* quit을 입력받을 때까지 계속 실행 */
35	while(isRun)
36	{
37	/* 통보문 읽기 */
38	result = msgrcv(msgQid, (void *)&userT, BUFLEN, msgType, 0);
39	if(result == -1)
40	{
41	printf("msgrcv실행 실패오유번호: %d\n",errno);
42	return 0;
43	}
44	/* 읽어들인 통보문 출력, quit가 있는지 검사 */
45	printf("<< 사용자 >>\n시민증번호: %s\n", userT.userNo);
46	printf("주소: %s\n", userT.address);
47	if(!strncmp(userT.userNo, "quit",4))
48	{
49	isRun = 0;
50	}
51	}
52	
53	/* 통보대기렬를 제거한다. */
54	result = msgctl(msgQid, IPC_RMID, 0);
55	if(result == -1)
56	{
57	printf("msgctl실행 실패\n");
58	return 0;
59	}
60	return 1;
61	}

이번에는 통보문전송을 목적으로 하는 sendMsg.c의 원천코드를 보자.

실례프로그람: sendMsg.c	
1	#include <stdio.h>

2	#include <string.h>
3	#include <errno.h>
4	
5	#include <sys/types.h>
6	#include <sys0/ipc.h>
7	#include <sys/msg.h>
8	
9	/* 통보문으로 주고 받을 자료의 길이와 형정의 */
10	#define BUFLEN 32
11	typedef struct
12	{
13	long int msgType; /* 통보문의 형 */
14	char userNo[13+1]; /* 사용자의 시민증번호 */
15	char address[17+1]; /* 사용자의 주소 */
16	} UserType;
17	
18	int main()
19	{
20	/* 필요한 변수선언 */
21	int isRun = 1;
22	int msgQid, result;
23	char buffer[17];
24	UserType userT;
25	
26	/* 통보대기렬의 ID 얻어오기 */
27	msgQid = msgget ((key_t) 8000, 0666 | IPC_CREAT);
28	if(msgQid == -1)
29	{
30	printf("msgget실행 실패오유번호: %d\n", errno);
31	return 0;
32	}
33	
34	/* quit을 입력받을 때까지 계속 실행 */

35	while(isRun)
36	{
37	/* 통보문 작성 */
38	memset(&userT, '\0', BUFLEN);
39	userT.msgType = 1;
40	printf("시민증번호:"); gets(buffer); strncpy(userT. userNo, buffer, 13);
41	printf("주소:"); gets(buffer); strncpy(userT.address, buffer, 17);
42	
43	/* 작성된 통보문을 전송 */
44	result = msgsnd(msgQid, (void *)&userT,BUFLEN,0);
45	if(result == -1)
46	{
47	printf("msgsnd실행 실패\n");
48	isRun = 0;
49	}
50	
51	/* 시민증번호에 quit가 포함되였는지 검사 */
52	if(!strncmp(userT.userNo, "quit", 4))
53	{
54	isRun = 0;
55	}
56	}
57	return 1;
58	}

프로그람원천코드의 작성 및 분석이 끝났으면 콤파일을 진행하고 실행을 시켜보자. 실행을 할 때에는 두개의 쉘창을 리용하여야 한다.(그림 6-4, 6-5)

그림 6-4. rcvMsg.c 의 실행결과

그림 6-5. sendMsg.c 의 실행결과

상식

콤퓨터망관리

콤퓨터망이 급속히 확대발전됨에 따라 망체계가 여러 회사들에서 생산한 제품들로 구성하고 있는 실정에서 오유가 발생하는 경우 그에 대응하기가 매우 어려워지게 된다. 이로부터 콤퓨터망의 관리를 일체화, 전문화해야 할 필요성이 제기되게 된다.

콤퓨터망관리에는 구성관리, 장애관리, 성능관리, 계좌관리, 보안관리 등 5 가지로 이루어진다. 관리를 위한 규약에는 SNMP(Simple Network Management Protocol)와 CMIP(Common Management Information Protocol)의 두가지가 있다.

제3절. C++ 언어를 리용하여 IPC를 실현하는 프로그람작성

지금까지는 C언어를 리용하여 IPC를 실현해보았다.

이 절에서는 C++언어를 리용하여 프로그람을 작성하는 방법을 실례프로그람을 작성하면서 설명한다. 이때는 통보대기렬을 리용하는 방법을 사용한다. 작성할 실례프로그람은 하나의 프로그람이 통보문전송과 처리를 담당할수 있다.

그리고 통보대기렬을 사용하는 모듈은 스레드로 움직이도록 작성한다. 스레드로 돌아가는 모듈을 리용하여 통보문의 전송과 처리가 중단되지 않게 만든다. 그러면 먼저 통보대기렬에서 사용될 자료형을 정의하고 클라스(class)를 설계하도록 하자. 아래의것은 통보대기렬를 리용하여 주고받을 자료구조체의 구조이다.

```
// 통보대기렬에서 사용할 자료의 길이와 형을 정의
#define DATALEN 32
typedef struct
{ --
long int msgType; /* 통보문형*/
char userNo[13+1]; /* 13 자리, 가입자의 주민등록번호 */
char password[8+1] ; /* 8 자리, 가입자가 등록한 통과어*/
char hpNo[8+1] ; °  /* 8 자리, 가입자의 전화번호 */
} DataType;
```

이제 클라스를 설계하여 보자. 클라스는 내부에 스레드를 위한 구조체로 private성원변수와 스레드를 실행시키는 함수, 스레드로 실행될 모듈들로 구성된다. 그리고 스레드들은 수신(receive)을 담당할 스레드와 전송(send)을 담당할 스레드로 나누어진다. 다음은 프로그람의 주클라스인 MsgQ의 사용형식을 보여주고있다.

```
class MsgQ
{
public:
      //readMsgQ 를 실행시키는 함수
      bool runReadMsgQ();
      //스레드로 돌아가는 함수. 대기렬로부터 자료를 읽어들인다.
      static void *readMsgQ(void *_arg);
      //writeMsgQ 를 실행시키는 함수
      bool runWritMsgQ();
      //스레드로 돌아가는 함수. 대기렬에 자료를 입력한다.
      static void *writeMsgQ(void *_arg);
      //Config 파일안에 있는 통보대기렬의 열쇠값을 얻는다.
int getQKey() ;
private:
      //read/Write 모듈들을 위한 스레드
      pthread_t readMsgQ_thread;
      pthread_t: writeMsgQ_thread;
};
```

그러면 이러한 사용형식들을 바탕으로 하여 작성된 전체 프로그람의 원천코드를 보도록 하자. 다음은 MsgQ프로그람의 머리부파일인 MsgQ.h파일의 원천코드이다.

실례프로그람: MsgQ.h	
1	#ifndef _MSGQ_H_
2	#define _MSGQ_H_
3	
4	// msgQ에서 사용할 자료의 길이와 형 정의
5	#define DATALEN 32
6	typedef struct
7	{
8	long int msgType; /* 통보문의 형 */
9	char userNo[13+1]; /* 13자리 (가입자의 시민증번호) */
10	char password[8+1]; /* 8자리 (가입자가 등록한 통과어) */
11	char hpNo[8+1] ; /* 8자리 (가입자의 전화번호) */
12	} DataType ;
13	
14	class MsgQ
15	{
16	public:
17	// readMsgQ를 실행시키는 함수
18	bool runReadMsgQ();
19	// 스레드로 돌아가는 함수. 대기렬로부터 자료를 읽어들인다.
20	static void *readMsgQ(void *_arg);
21	
22	// writeMsgQ를 실행시키는 함수
23	bool runWriteMsgQ();
24	// 스레드로 돌아가는 함수. 대기렬에 자료를 입력한다.
25	static void *writeMsgQ(void *_arg);
26	
27	// config 파일속에 있는 msgQ의 열쇠값을 얻는다.
28	int getQKey();
29	
30	private :

31	// read/write 모듈들을 위한 스레드
32	pthread_t readMsgQ_thread;
33	pthread_t writeMsgQ_thread;
34	};
35	
36	#endif /* _MSGQ_H_ */

이번에는 MsgQ.h파일을 사용하는 MsgQ.cpp 파일의 원천코드를 보도록 하자.

실례프로그람: MsgQ.cpp	
1	#include <iostream.h>
2	
3	#include <stdio.h>
4	#include <string.h>
5	#include <errno.h>
6	#include <pthread.h>
7	
8	#include <sys/types.h>
9	#include <sys/ipc.h>
10	#include <sys/msg.h>
11	
12	#include "MsgQ.h"
13	
14	/***
15	* FUNCTION : runReadMsgQ
16	* DESCRIPTION : msgQ의 열쇠값을 얻은후 스레드를 실행시키는 함수
17	***/
18	bool
19	MsgQ::runReadMsgQ()
20	{
21	int ret;
22	// msgQ의 열쇠를 얻은 후 void* 형으로 변환
23	int QKey = getQKey();
24	if(QKey <= 0)
25	{

26	cout << "MsgQ::readHlrMsgQ() QKey 값 이상" << endl;
27	return false;
28	}
29	void *_arg = (void *)QKey;
30	
31	// void*형으로 변환한 msgQ의 열쇠값과 함께 스레드함수 실행
32	if(ret = pthread_create(&readMsgQ_thread, NULL, MsgQ::readMsgQ, _arg))
33	{
34	cout << "readMsgQ 스레드의 실행에 실패:" << strerror(ret) << endl;
35	return false;
36	}
37	return true;
38	}
39	
40	/* *
41	* FUNCTION : readMsgQ
42	* DESCRIPTION : 스레드로 실행될 static로 선언된 방법. 통보문을 읽는다.
43	* ** * * * * * * * * * * * * * */
44	void*
45	MsgQ::readMsgQ(void *_arg)
46	{
47	// void*로 입력된 열쇠값을 int형으로 변환한다.
48	int QKey = (int)_arg;
49	int msgQid;
50	DataType dataT;
51	long int msgType = 0;
52	bool isRun = true;
53	
54	// 열쇠값을 리용하여 msgQ 생성 및 msgQ 얻기
55	msgQid = msgget((key_t) QKey, 0666 | IPC_CREAT);
56	if(msgQid == -1)
57	{

58	cout << "MsgQ: :readMsgQ msgget실행실패" << endl;
59	return NULL;
60	}
61	
62	// 통보문을 읽고 화면에 출력, 시민증번호가 quit이면 완료
63	while(isRun)
64	{
65	if(msgrcv(msgQid, (void *)&dataT, DATALEN, msgType, 0) == -1)
66	{
67	cout << "MsgQ :: readMsgQ msgrcv실행실패!" << endl;
68	isRun = false;
69	}
70	if(!strncmp(dataT.userNo, "quit", 4))
71	isRun = false;
72	cout << "<< 가입자 정보>>" << endl;
73	cout << "시민증번호: " << dataT.userNo << endl;
74	cout << "통과어: " << dataT.password << endl;
75	cout << "전화번호: " << dataT.hpNo << endl;
76	}
77	
78	// msgQ를 핵심부에서 제거한다.
79	if(msgctl(msgQid, IPC_RMID, 0) == -1)
80	{
81	cout << "MsgQ::readMsgQ msgctl 실행실패!" << endl;
82	return NULL;
83	}
84	
85	return NULL;
86	}
87	
88	/* *
89	* FUNCTION : runWriteMsgQ
90	* DESCRIPTION : msgQ의 열쇠값을 얻은 후 스레드를 실행시키는 함수

91	* ** * * * * * * * * * * * * * * * * */
92	bool
93	MsgQ::runWriteMsgQ()
94	{
95	int ret;
96	// msgQ의 열쇠를 얻은 후 void* 형으로 변환
97	int QKey = getQKey();
98	if(QKey <= 0)
99	{
100	cout << "MsgQ::writeMsgQ() QKey 값이상!" << endl;
101	return false;
102	}
103	void *_arg = (void *)QKey;
104	
105	// void* 형으로 변환한 msgQ의 열쇠값과 함께 스레드함수 실행
106	if(ret = pthread_create(&writeMsgQ_thread, NULL, MsgQ::writeMsgQ, _arg))
107	{
108	cout << "writeMsgQ 스레드실행실패: " << strerror(ret) << endl;
109	return false;
110	}
111	return true;
112	}
113	
114	/* *
115	* FUNCTION : WriteMsgQ
116	* DESCRIPTION : 스레드로 실행될 static로 선언된 방법. 통보문을 적는다.
117	* ** * * * * * * * * * * * */
118	void*
119	MsgQ::writeMsgQ(void *_arg)
120	{
121	// void*로 입력된 열쇠값을 int형으로 변환한다.
122	int QKey = (int)_arg;
123	bool isRun = true;

124	int msgQid;
125	DataType dataT;
126	char buffer[14];
127	
128	// 열쇠값을 리용하여 msgQ생성 및 msgQ얻기
129	msgQid = msgget((key_t)QKey, 0666 | IPC_CREAT);
130	if(msgQid == -1)
131	{
132	cout << "MsgQ::readMsgQ msgget 실행실패!" << endl;
133	return NULL;
134	}
135	
136	// 통보문을 작성한후 대기렬에 쓰기한다.
137	while(isRun)
138	{
139	memset (&dataT, '\0', DATALEN);
140	dataT.msgType = 1;
141	cout << "시민증번호[13]:";gets(buffer);
142	strncpy(dataT.userNo,buffer,13);
143	cout << "통과어[8] :";gets(buffer);
144	strncpy(dataT. password,buffer, 8);
145	cout << "전화번호[8]: " ; gets(buffer);
146	strncpy(dataT.hpNo,buffer,8);
147	if(msgsnd(msgQid, (void *)&dataT, DATALEN, 0) == -1)
148	{
149	cout << "MsgQ::writeMsgQ msgsnd 실행실패!" << endl;
150	isRun = false;
151	}
152	// 사용자 시민증번호에 quit가 입력되면 완료한다.
153	if(!strncmp(dataT.userNo,"quit",4))
154	isRun = false;
155	}
156	return NULL;
157	}

```
158
159  /* * * * * * * * * * * * * * * * * * * * * * * * * *  * * * * * * * * * * * * *
160  * FUNCTION : getQKey
161  * DESCRIPTION : Config 파일로부터 msgQ의 열쇠값을 받아오는 함수
162  * * * * * * * * * * * * * * * * * * * * * * * * * ** * * * * * * * * * * * * */
163  int
164  MsgQ::getQKey()
165  {
166          return 9999;
167  }
168
169  /* * * * * * * * * * * * * * * * * * * * * *  * * * * * * * * * * * * * * * * * *
170  * FUNCTION : main
171  * DESCRIPTION : 객체 생성 및 실행. Sender 또는 receiver로 설정 가능
172  * * * * * * * * * * * * * * * * * * * * * * ** * * * * * * * * * * * * * * * * */
173  int main(int argc, char* argv[ ])
174  {
175          // 인수의 개수 검사
176          if(argc !=2)
177          {
178                  cout << "Usage: MsgQ <sndQ | rcvQ>" << endl << endl;
179                  return 0;
180          }
181
182          // MsgQ객체 생성
183          MsgQ *msgQ = new MsgQ();
184
185          // 인수가 sndQ이면 write용스레드 실행
186          if(!strncmp(argv[1], "sndQ",4))
187          {
188                  if(msgQ->runWriteMsgQ() == false)
189                          return 0;
190          }
```

191	// 인수가 rcvQ이면 읽기용 스레드 실행
192	else if(!strncmp(argv[1], "rcvQ",4))
193	{
194	if(msgQ->runReadMsgQ() == false)
195	return 0;
196	}
197	else
198	{
199	cout << "Usage: MsgQ <sndQ | rcvQ>" << endl << endl;
200	delete(msgQ);
201	return 0;
202	}
203	// 주스레드 완료
204	pthread_exit(0);
205	return 1;
206	}

상식

쉘

UNIX와 대화하기 위한 지령해석프로그람으로서 UNIX의 지령을 해석하고 실행하는 번역기이다. 조작체계의 제일 바깥쪽에 있으며 조작체계전체를 둘러싸고있는듯한 느낌으로 되여있기때문에 쉘(껍질)이라는 이름을 달았다. 쉘지령을 사용하여 작성한 프로그람을 쉘 Script라고 부르며 지령처럼 실행시킬수 있다. if~then~else와 같은 조종문이나 변수도 사용할수 있으며 간단한 프로그람을 작성할수도 있다. 쉘은 여러 종류가 개발되여있는데 Bourne 쉘, C 쉘 등이 보급되고있다.

제7장

체계간 통신 (1)

서론

7장과 8장에서는 먼거리에 떨어진 체계들사이의 통신프로그람을 작성하기 위한 내용들을 배우게 된다. 여기서 사용되는 방법은 소케트를 리용한것으로써 Berkeley Socket를 기본으로 한다. 7장에서는 먼저 소케트통신과 관련된 전반적인 내용을 취급한다. 그 다음 소케트프로그람작성에서 사용하는 체계호출함수에 대해 설명한다. 그리고 이것을 리용한 간단한 소케트프로그람을 작성하게 된다.

소케트프로그람작성에서는 프로그람작성과정을 기본으로 보여주는데 이 내용은 다음 장에서 작성하는 프로그람들의 기초코드로 활용된다. 7장의 차례는 다음과 같다.

목표

1. 소케트통신
2. 소케트체계호출함수
3. 소케트프로그람작성

제1절. 소케트통신

이 절에서는 소케트통신을 위한 기본 API를 소개하고 간단한 TCP/IP프로그람에 대해 설명한다. 이때 사용하는 소케트프로그람은 Berkeley Socket를 기본으로 하고있다.

7.1.1. 파일입출구(File I/O)와의 비교

먼저 소케트통신과 파일 I/O를 비교해 보자. 기본적인 파일I/O함수는 open, create, close, read, write, lseek 등을 리용한다. 하나의 처리기가 파일에 무엇인가를 남기고 다른 처리기가 이 파일을 읽어들인다고 가정해보면 파일에 무엇인가를 남기는 처리기는 파일이 등록부안에 존재하는가를 확인하고 없으면 새로 생성(create)한다.

만일 이미 존재하던 파일이면 파일을 열고 파일에서 요구하는 내용을 써넣고 마지막으로 파일을 닫는다. 이와 같은 봉사를 하는것을 소케트통신에서 봉사기라고 생각하면 된다. 그러면 파일의 읽기를 원하는 처리기는 똑같은 순서로 파일에서 자료를 읽어들이게 된다.

이와 비슷한 역할을 하는것을 소케트통신에서는 자료를 받아들이는 의뢰기라고 보면 된다. 하지만 소케트통신은 파일입출력과는 다른데 그 차이는 다음과 같이 볼수 있다.

- 소케트에서 의뢰기와 봉사기관계는 서로 대칭적이 아니다. 봉사기는 봉사를 제공해주는 립장에서 의뢰기의 요구를 받아들이도록 언제나 준비를 하고있고 의뢰기는 봉사를 요청하기 위한 준비를 한다.
- 콤퓨터망의 련결은 파일입출력처럼 언제나 련결되여있는 방식(Connection Oriented)과 규약에 따라 자료를 주고받을 때만 잠시 련결되는 방식(Connection Less) 두가지로 나누어 본다. Connection Less방식은 콤퓨터망상의 각종 다른 주콤퓨터(Host)와 련결되기때문에 열기(open)와 같은 함수가 존재하지 않는다
- 콤퓨터망상의 I/O는 파일의 I/O에 비해 그 이름이 중요한 역할을 한다. 파일 I/O는 처음 파일이 권한에 따라 열려지면 그 권한에 따라 모든 역할이 규정되지만 콤퓨터망의 I/O는 접속되는 상대의 이름에 따라 각기 다른 권한을 가질수 있다.
- 콤퓨터망의 I/O는 좀 더 복잡한 규약과 파라메터를 가진다. (Protocol, local-addr, local-process, foreign-addr, foreign-processl) 그 리유는 콤퓨터망의 I/O가 파일 I/O에 비해 보다 복잡하고 정교하기때문이다.
- 콤퓨터망의 I/O는 다중사용자통신(Multiple Communication Protocol)을 지원한다. 즉 한가지 정해진 일만 하는것이 아니라 규약에 따라 또는 그 봉사에 따라 여러 사용자들에게 동시에 봉사를 제공한다.

그러면 표 7-1를 리용하여 콤퓨터망의 I/O와 파일의 I/O를 비교해보자.

표 7-1. 콤퓨터망 I/O(Socket)와 파일 I/O 비교

구분	동작	Socket	File I/O
Server (봉사기)	창조	Socket()	Open()
	묶기주소	Bind()	
	대기렬기록	Listen()	
	련결대기	Accept()	
Client (의뢰기)	창조	Socket()	Open()
	묶기주소	Bind()	
	봉사기접속	Connect()	
	자료처리	Read()	
		Write()	Write()
		Recv()	Recv()
		Send()	
	데이터그램	Recvfrom()	
		Sendto()	
	완료	Close()	Close()
		Shutdown()	Unlink()

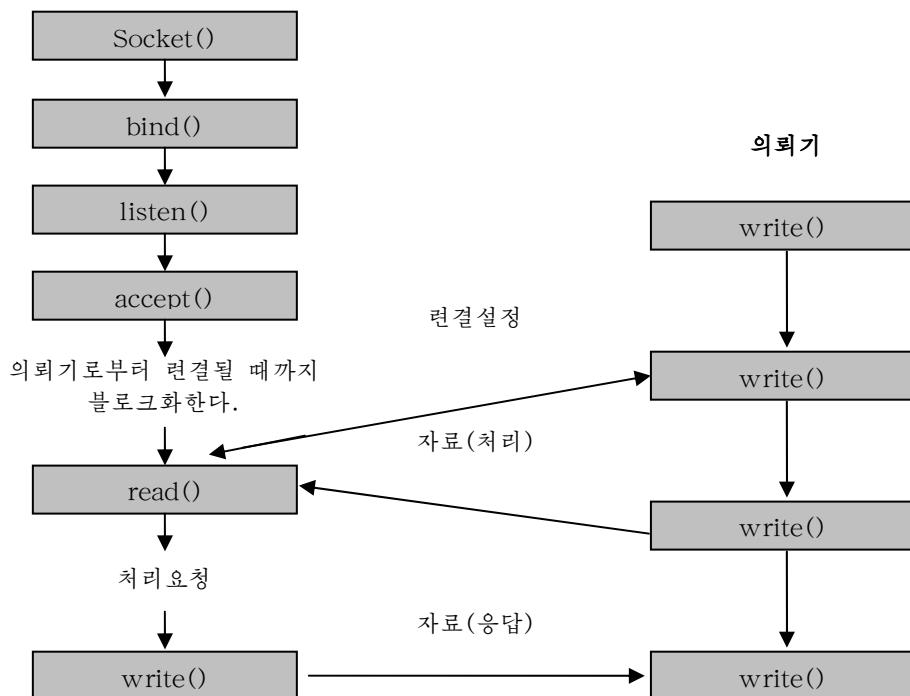



그림 7-1. 련결형소케트 프로그람의 흐름

7.1.2. 실행과정

그림 7-1은 일반적으로 사용하는 련결형(Connection-Oriented)소케트프로그람의 흐름이다. 먼저 봉사기가 소케트를 열고 봉사준비를 하고있으면 의뢰기가 봉사기에 접속을 하고 봉사를 요청한다.

그림 7-2는 비련결형규약(Connection less Protocol)을 리용한 의뢰기-봉사기환경의 체계흐름이다. 의뢰기는 봉사기와 련결될 때 sendto를 리용하여 데이터그램과 파라메터를 봉사기에 보내고 봉사기 또한 의뢰기로부터 련결설정을 맺는 대신 recvfrom을 리용하여 자료가 오기를 기다린다.

recvfrom은 의뢰기로부터 데이터그램뿐만 아니라 의뢰기의 주소를 같이 받아들여 봉사기가 다시 의뢰기에 요청한 자료를 보낼 때 사용한다.

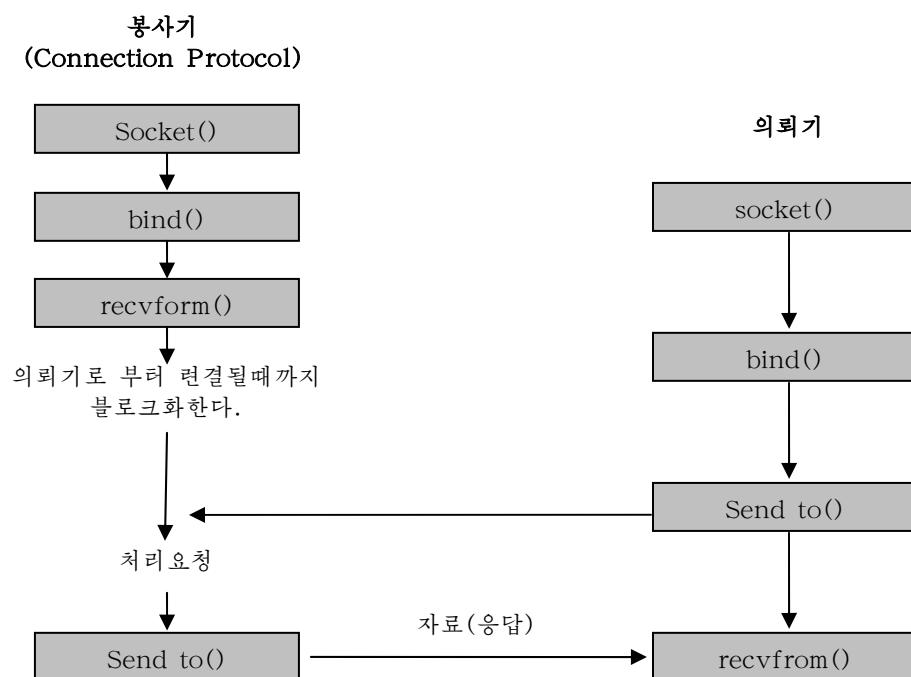



그림 7-2. 비련결형규약을 리용한 체계흐름

제2절. 소케트체계호출

대부분의 콤퓨터망프로그람에서 사용하는 구조는 다음과 같은 소케트구조체의 지적자(point)를 활용한다.

원천코드:

```
1  /* Structure used by kernel to store most addresses. */
2  struct sockaddr
3  {
4          u_short sa_family;   /* address family */
5          char sa_data[14] ;   /* up to 14 bytes of direct address */
6  };
7  /* internet address (old style || should be updated) */
8  struct in_addr {
9          union {
10                 struct {u_char s_b1,s_b2,s_b3,s_b4;}S_un_b;
11                 struct {u_short s_w1,s_w2; } S_un_w;
12                 u_long S_addr;
13         } S_un;
14 #define s_adder S_un.S_addr      /* can be used for most tcp & ip code */
15 #define s_host S_un.S_un_b.s_b2 /* host on imp */
16 #define s_net S_un.s_un_b.s_b1   /* network */
17 #define s_imp S_un.S_un_w.s_w2 /* imp */
18 #define s_impno S_un.S_un_b.s_b4        /* imp# */
19 #define s_lh S_un.S_un_b.s_b3    /* logical host */
20 };
21
22 /* Socket address, internet style. */
23 struct sockaddr_in {
24         short sin_family;
25         u_short sin_port;
26         struct in_addr sin_addr;
27         char sin_zero[8];
28 };
```

례를 들어 령역을 지원하는 콤퓨터망함수인 connect(혹은 bind)는 그 인자로서 소케트주소구조체(sockaddr)와 규약의 크기를 전달해주어야 한다. 즉 connect()함수의 경우에는 다음과 같이 사용한다.

```
  struct sockaddr_in serv_addr;
  int     connect(sockfd,     (struct     sockaddr*)&serv_addr,     sizeof
(serv_addr) );
```

우의 함수에서 두번째 인자는 인터네트주소의 지적자이고 세번째 인자는 인터네트주소의 크기이다. 이때 세번째 인자로 그 크기를 전달해주어야 하는 리유는 connect()함수 하나를 사용하여 XNS 등 다른 규약도 인자로 사용할수 있도록 하기 위해서이다. 이번에는 콤퓨터망프로그람에 필요한 기본 API에 대해 보도록 하자.

7.2.1. socket

콤퓨터망프로그람을 작성하기 위해서 반드시 알아야 하는 체계함수가 바로 socket() 체계호출함수이다. socket()함수를 통하여 사용하려고 하는 소케트의 규약을 지정해준다.

```
int socket(int family, int type, int protocol);
```

사용가능한 family류형은 20가지이상이지만 그중 많이 사용하는 형식들은 아래와 같다.

```
/ * local to host(pipes, portals) * /
#define AF_UNIX 1

/ * internetwork: UDP, TCP, etc. * /
#define AF_INET 2

/ *arpanet imp addresses * /
#define AF_IMPLINK 3

/ * IPX and SPX* /
#define AF_IPX 6

/ *XEROX NS protocols * /
#define AF_NS 6
```

우에서 AF_는 address family(주소계렬)의 첫 글자를 딴것이고 다른 류형으로 protocol family(규약계렬)를 나타내는 PF로 시작하는 PF_UNIX, PF_INET, PF_IMPLINK 등도 사용가능하다.

우에서 IMP는 Interface Message Processor(통지문처리기대면부)의 략자이며 ARPANET에서 사용하는 지능망파케트마디로서 초기 인터네트판본에서 사용하던것이지만 지금은 그리 흔하게 사용되지 않는다. 다음은 socket의 류형이다.

```
#define SOCK_STREAM1
/* stream socket * /
#define SOCK_DGRAM 2
/ * datagram socket * /
#define SOCK_RAW 3
/ * raw-protocol interface * /
#define SOCK_RDM 4
/ * reliably-delivered message */
#define SOCK_SEQPACKET 5
/ * sequenced packet stream * /
```

모든 류형의 socket와 family의 조합을 사용할수는 없다. 표 7-2에서 사용가능한 류형의 socket와 family의 조합을 보여주었다.

표 7-2. Socket와 family의 조합

조합	AF_UNIX	AF_INET	AF_NS
SOCK_STREAM	사용가능	TCP	Spp
SOCK_DGRAM	사용가능	UDP	사용가능
SOCK_RAW	사용불가능	IP	사용가능
SOCK_SEQPACKET	사용불가능	사용불가능	Spp

소케트프로그람을 작성하기 위해서는 우의 사용가능한 조합을 선택해서 사용해야 한다. protocol인자에는 거의 대부분의 응용프로그람이 0을 사용한다. 하지만 특수한 목적으로만 사용하기 위해서 가능한 조합을 만들어 사용할수 있다. 표 7-3에서는 사용가능한 조합을 보여주었다.

표 7-3. 규약과 사용가능한 조합

계렬	형식	규약	실지규약
AF_INET	SOCK_DGRAM	IPPROTO_UDP	UDP
AF_INET	SOCK_STREAM	IPPROTO_TCP	TCP
AF_INET	SOCK_RAW	IPPROTO_ICMP	ICMP*
AF_INET	SOCK_RAW	IPPROTO_RAW	(raw)
AF_NS	SOCK_STREAM	NSPROTO_SPP	SPP
AF_NS	SOCK_SEQPACKET	NSPROTO_SPP	SPP
AF_NS	SOCK_RAW	NSPROTO_ERROR	Error protocol
AF_NS	SOCK_RAW	NSPROTO_RAW	(raw)

우의 조합을 리용하여 다양한 응용프로그람을 만들수 있다. 그중에서 가장 대표적인 것이 바로 ICMP를 사용하는 Ping프로그람이다.

Socket함수는 파일입출력함수와 같이 16bit정수를 반환(return)한다. 앞으로는 이 반환값을 socketfd라고 부른다.

socketfd와 관련된 내용을 보면 protocol, local-addr, local-process, foreign-addr, foreign-process 등이 있고 처리기가 의뢰기로서 역할을 하는가 봉사기로서 역할을 하는가에 따라 함수를 사용하는 방법이 달라진다. 또한 련결설정규약(Connection Oriented Protocol)을 사용하는가 혹은 비련결설정규약(Connection less protocol)을 사용하는가에 따라 다른 함수의 조합을 사용하게 된다.

대표적인것이 system follow이다.(표 7-4)

표 7-4. system follow

구 분	규 약	국부주소 국부처리	외부주소 외부처리
련결형봉사기	Socket( )	Bind( )	Listen( )
련결형의뢰기	Socket( )	connect( )	accept( )
비련결형봉사기	Socket( )	Bind( )	Recvfrom( )
비련결형의뢰기	Socket( )	Bind( )	Sendto( )

7.2.2. bind

bind()함수는 아직 이름이 붙여지지 않은 socket에 이름을 명명하는것으로서 사용형식은 다음과 같다.

```
int bind(int sockFd, struct sockaddr * addr, int namelen);
```

bind()함수의 두번째 인자는 자기주소에 대한 지적자이고 세번째 인자는 자기주소의 크기이다.

봉사기는 bind를 통해 체계의 알려진 주소를 등록한다. 이렇게 함으로써 알려진 주소로 전달되는 어떤 통보문이든 받을 준비를 하는것이다. 련결설정봉사기나 비련결설정봉사기는 모두 의뢰기의 요청을 받아들이기전에 반드시 bind를 수행해야 한다.

의뢰기의 경우에는 의뢰기가 스스로 자기의 주소를 등록한다. 비련결설정의뢰기는 자기의 고유주소를 bind()해주어야 하는데 그렇지 않으면 봉사기는 그 요청을 받아들이더라도 어디서 요청을 받았는지를 모르기때문에 정상적인 통신을 진행할수 없다.

7.2.3. connect

의뢰기처리기는 봉사기와 련결설정을 하기 위해 socket를 socketfd에 련결한다. 앞에서도 설명한것처럼 sockfd는 socket()함수가 반환한 값이다.

```
int connect (int sockFd, struct sockaddr *name, int namelen);
```

connect()함수의 두번째 인자는 소케트주소에 대한 지적자이고 세번째 인자는 소케트주소의 크기이다. 대부분의 련결(Connection_oriented)규약은 connect함수를 사용함으로써 외부체계와 자기와의 련결을 실현한다. 이 함수는 내부적으로 두 체계간의 통보문을 교환하여 련결이 확실히 이루어진 다음에야 련결설정에 대한 성공값을 반환한다.

의뢰기는 일반적으로 connect()함수를 실행하기전에 련결할 필요는 없다. 비련결의뢰기도 connect를 사용해야 하지만 련결설정의뢰기에서 의미하는 connect의 의미와는 다르다.

비련결의뢰기는 상대방의 알려진 주소를 보관하고있다가 통신이 이루어지면 socket계층에서 봉사기에 자료를 전송한다. 그러므로 비련결의뢰기는 매번 봉사기에 자료를 전송할 때마다 봉사기와 련결할 필요가 없다.

7.2.4. listen

listen()함수는 련결(Connection oriented)봉사기에서 의뢰기의 련결을 기다리게 하는 역할을 하는 함수로서 다음과 같은 사용형식을 가진다.

```
int listen(int sockFd, int backlor);
```

이 함수는 대개 socket()와 bind()를 실행한 다음에 그리고 accept()를 실행하기전에 사용한다. 두번째 인자인 backlog는 체계가 얼마나 많은 의뢰기를 받아들이겠는가에 대한 대기렬을 설정하는것이다. 이 인자는 보통 5로 설정을 하며 체계의 성능에 따라 그 크기를 다르게 설정하여 준다.

7.2.5. accept

accept()함수는 련결봉사기(connection oriented Server)가 우에서 설명한 listen()함수를 수행한후 의뢰기로부터 련결설정을 기다리고있다가 실제 련결요청이 들어오면 봉사기가 그 련결을 받아들이라는 지령이다.

```
int accept(int sockFd, struct sockaddr * addr, int * addrlen);
```

accept()함수는 대기렬에 있는 첫번째 요청을 받아들여 련결을 하고 다시 socketfd와 꼭같은것을 만들어 대기렬에 있는 다음 련결요청을 받아들이도록 만든다.

두번째 인자인 addr은 connect()함수에서 반환된 주소이며 세번째 인자인 addrlen은 그 주소의 길이이다.

이 함수의 반환값은 의뢰기와 통신한 새로운 socketfd이고 의뢰기와의 통신은 새로운 socketfd을 통해서 이루어지게 함으로써 봉사기는 언제나 같은 포구를 열고 의뢰기의 련결을 기다리게 된다. 다음의 실례는 전형적이라고 볼수 있는 원천코드이다.

원천코드:

```
1   int socketfd, newsocketfd;
2   
3   s = socket(. . . . . .);
4   if (s < 0)
5           Error_Message("소케트 생성 실패");
6   
7   err = bind(s, (struct sockaddr *)&localAddr, sizeof(localAddr));
8   if(err < 0)
9           Error_Message("소케트의 련결에 실패");
10  
11  err = listen(s, 5);
12  if(err < 0)
13          Error_Message("소케트의 요청에 실패");
14  
15  while(TRUE)
16  {
17          newsockfe = accept(s, . . . . .);
18          if(newsockfd < 0)
19                  Error_Message("소케트 접수오유");
20  
21          _beginthread(start_address, 0,(void *)newsockfd);
22  }
```

즉 봉사기는 의뢰기로부터 련결요청을 받아들이고 봉사를 하기 위한 스레드를 생성하는데 그 인자로서 newsocketfd를 전달해준다. accept()함수에 의하여 반환된 newsockfd는 규약, 자기 주소, 자기 프로쎄스, 상대방의 주소, 상대방의 프로쎄스 등의 내용을 모두 가지고있으며 원래의 socketfd는 상대방의 주소와 상대방의 프로쎄스를 제외한 3개만 가지고 다음 의뢰기의 련결요청을 기다리게 된다.

우리는 지금까지의 설명을 통하여 봉사기가 여러개의 의뢰기와 동시에 통신을 하기 위해서 다른 소케트를 임의로 만들지 않아도 accept()함수가 이런 역할을 대신해 준다는것을 알수 있다. 또한 봉사기의 대기렬에 봉사를 할 내용들이 많다하더라도 하나씩 순서대로 처리하기를 요구한다면 스레드를 생성하지 않고 프로쎄스에서 봉사를 차례로 하게 할수 있다는것도 알수 있다.

원천코드:

```
1   int socketfd, newsocketfd;
2   
3   s = socket(. . . . . .);
4   if (s < 0)
5           Error_Message("소케트 생성 실패");
6   
```

7	err = bind(s, (struct sockaddr *)&localAddr, sizeof(localAddr));
8	if(err < 0)
9	Error_Message("소케트의 련결에 실패");
10	
11	err = listen(s, 5);
12	if(err < 0)
13	Error_Message("소케트의 요청에 실패");
14	
15	while(TRUE)
16	{
17	newsockfe = accept(s, ...);
18	if(newsockfd < 0)
19	Error_Message("소케트 접수오유");
20	Do_SerVice(newsockfd);
21	close(newsockfd) ;
22	}

7.2.6. 통보문의 송수신을 위한 체계호출함수

통보문을 송수신하는 함수들로는 read(), write(), recvfrom() 그리고 sendto() 등이 있다. 이러한 함수의 사용형식을 보면 다음과 같다.

```
int send(int sockFd, char * buf, int len) ;
int Sendto (int sockFd, char * buf, int len, int flags, struct sockaddr
* to, int tolen) ;
int read(int sockFd, char * buf, int len) ;
int recvfrom(int sockFd, char * buf, int len, int flags, struct sockaddr
* from, int * fromlen) ;
```

처음 세개의 인자 sockFd, buf, len은 소케트로부터 입출력할 완충기와 그 크기를 지정한 인자들이 된다. flags라는 인자는 대체로 0을 사용하지만 특수한 목적에 따라 다음과 같은 값을 설정해주기도 한다.

```
#define MSG_OOB 0x1 /* process out-of-band data * /
#define MSG_PEEK 0x2 /* peek at incoming message (recvfrom 에서) */
#define MSG_DONTROUTE 0x4 /* send without using routing tables */
```

sendto()함수에서 to라는 인자는 자료를 보내야 할 주소를 가리키는 지적자이다. 여기서 sendto()함수는 규약에 의존하기때문에 콤퓨터망의 하위계층에서 인식할수 있도록 그 주소의 크기를 설정해주어야 한다. recvfrom()함수 역시 sendto와 같은 리유로 from과 len의 값을 설정해주어야 한다. sendto()함수의 마지막 인자는 정수형이지만 recvfrom()의 마지막 인자는 정수형지적자라는 점에서 주의하여 사용하여야 한다.

우의 네개 함수들은 파일입출력(파일 I/O)함수와 마찬가지로 보내고 받은 길이를 반환하여 준다. 비련결과 함께 사용되는 recvfrom(), sendto()함수는 전송받은 데이터그램의 크기를 반환하여준다.

7.2.7. close

처리기에서 사용한 소케트를 마지막에는 닫아주고 프로세스를 완료하여야 하는데 이때 사용하는 함수가 close이다. close()함수의 사용형식은 다음과 같다.

```
int close(int sockFd);
```

소케트가 닫혀있다는 의미는 체계핵심부안의 모든 자료가 전송되였다는것을 나타낸다. 하지만 close()함수를 사용하여 소케트를 닫으라는 명령을 체계에 주어도 이미 보내도록 대기렬에 전송된것들은 핵심부에 의해 여전히 상대처리기에 전달될 때까지 계속 전송을 시도할수 있다. 이러한 현상을 방지하기 위해서는 소케트추가선택(socket option)의 SO_LINGER를 사용하여 대기렬에 쌓인 자료를 반드시 전송하고 이미 전송된 자료는 핵심부에서 지워지도록 해주어야 한다.

7.2.8. 바이트순서 변환

다음 네개의 함수는 다른 콤퓨터체계에서 콤퓨터망의 바이트순서를 변환하는 함수로서 인터네트규약을 기준으로 만들어졌다.

```
u_long htonl(u_long hostlong) ; // Convert host-to-network;
long integer
u_Short htons(u_short hostshort); // Convert host-to-network;
short integer
u_long ntohl (u_long netlong); // convert network-to-host;
long integer
u_Short ntohS(u_Short netshort); // Convert network-to-host;
short integer
```

7.2.9. 주소변환

인터네트주소는 192.168.0.101처럼 4개의 점(.)을 포함한 주소로 만들어져있지만 실지로 콤퓨터가 인식하는 인터네트주소는 in_addr의 형태로 만들어져있다. 따라서 우리는 사람의 눈으로 확인할수 있는 인터네트주소의 형식을 콤퓨터가 인식할수 있는 인터네트주소의 형식으로 바꾸기도 하고 그 반대로 바꾸어주기도 하여야 한다.

```
unsigned long inet_addr (const char * ) ;
char * inet_ntoa(struct in_addr in) ;
```

inet_addr는 사람이 인식할수 있는 형태의 인터네트주소를 콤퓨터가 인식할수 있는 형태로 바꾸어준다. 그 반대로 inet_nto는 콤퓨터가 인식할수 있는 주소를 사람이 인식할

수 있는 형태의 주소로 바꾸어주는 함수이다.

원천코드:	
1	bzero((char *) (&remoteAddr, sizeof (remoteAddr)) ;
2	remoteAddr.sin_family = AF_INET;
3	remoteAddr.sin_addr.s_addr=inet_addr (server_name) ;
4	remoteAddr.sin_port = htons(port) ;

우의 실례는 인터네트주소를 콤퓨터가 인식할수 있는 형태의 인터네트주소로 바꾸는 것을 보여준것이다. 별명(alias)형태의 주소를 콤퓨터가 인식할수 있는 주소로 바꾸어주는 함수는 gethostbyname이다. 또한 DNS(Domain Name Server)가 설정되여있는 콤퓨터에서는 DNS를 통한 주소변환도 가능하다.

원천코드:	
1	Localhost
2	Young
3	Mountain
4	doctor.co.kp

```
struct hostent * gethostbyname(const char * name) ;
```

다음은 gethostbyname()을 통한 주소변환의 실례이다.

원천코드	
1	Char server_name[16] = "doctor.co.kp";
2	struct sockaddr_in remoteAddr;
3	struct hostent * phe;
4	
5	if((phe = gethostbyname(server_name)) !=NULL)
6	{ memcpy((char *)&(remoteAddr.sin_addr), phe->h_addr ,phe->_length);
7	}

7.2.10. 소케트추가선택항목을 위한 함수

다음은 소케트의 추가선택항목과 관련된 기능을 제공하는 함수들의 사용형식이다.

```
int setsockopt (int SOCkFd, int level, int optname, const char *optval, int optlen);
int getsockopt (int sockFd, int level, int optname, char * optval, int FAR * optlen);
```

두번째 파라메터인 level은 어떤것이 추가선택항목을 새치기하는가를 나타낸다. 일반적으로 TCP/IP나 XNS에서 많이 사용하지만 추가선택항목에 따라 모든 소케트에 적용되기도 한다. 세번째 파라메터인 optval은 사용자정의변수의 지적자로서 getsockopt 함수에서는 체계로부터 반환되는 값이고 setsockopt에서는 체계에서 설정하는 값이다.

무엇보다 중요한것은 이 값이 문자형지적자로 이루어졌다는것이다. 마지막 인자인 optlen은 변수의 크기와 관련된 인자이다. 추가선택항목에는 두가지 류형이 있다. 하나는 추가선택항목을 실행하거나 또는 실행하지 않게 하는 flag라는 추가선택항목과 다른 하나는 체계에 값을 설정하거나 얻으려고 하는 체계정보를 얻을수 있는 value이다. 다음 표에서 flag에 0이 설정되여있으면 flag를 의미하는것이고 그렇지 않으면 value를 의미하는것이다.

표 7-5. flag와 value

준위	Optname	Get값	Set값	설명	flag	자료형
IPPROTO_IP	IP_OPTIONS	0	0	IP머리부추가선택		
IPPROTO_TCP	TCP_MAXSEG	0		TCP최대토막달기		Int
	TCP_NODELAY	0	0	파케트를 모으지 않고 바로 전송	0	Int
SOL_SOCKET	SO_BROADCAST	0	0	통보문방송을 허용	0	Int
	SO_DEBUG	0	0	오유해석기설정	0	Int
	SO_DONROUTE	0	0	대면부주소만 사용		Int
	SO_ERROR	0		오유의 종류를 알아내고 제거	0	Int
SOL_SOCKET	SO_KEEPALIVE	0	0	련결이 중간에 끊기지 않도록 설정	0	int
SOL_SOCKET	SO_LINGER	0	0	자료가 존재하면 닫기전에 잠시 대기		int
	SO_OOBLINE	0	0			int
	SO_RCVBUF	0	0	전달될 완충기의 크기		int
	SO_SNDBUF	0	0	전송될 완충기의 크기		int
	SO_RCVLOWAT	0	0	Low-water 표시 전달받음	0	int
	SO_SNDKOWAT	0	0	Low-water 표시 전송		int
	SO_RCVTIMEO	0	0	전달될 제한시간	0	int
	SO_SNDTIMEO	0	0	전송될 제한시간		int
	SO_REUSEADDR	0	0	주소 재사용 허용	0	int
	SO_TYPE	0		소케트의 류형을 얻는다		int
	SO_USELOOPBACK	0	0		0	int

아래의 원천코드는 추가선택항목을 체계에 설정하고 얻을수 있는 간단한 실례이다.

원천코드:	
1	int ReceiveBufferSize, SendBufferSize;
2	int sockFd = socket (AF_INET, SOCK_STREAM, 0);
3	if(sockFd < 0) {
4	printf("Failed to create socket\n");
5	Return;
6	}
7	
8	// 전달받을수 있는 완충기의 크기를 구한다.
9	int Err = setsockopt(socketfd, SOL_SOCKET, SO_RCVBUF, (char *)&
10	ReceiveBufferSize, sizeof(ReceiveBufferSize));
11	
12	// 전송완충기의 크기를 설정
13	SendBufferSize = 16384;
14	err = setsockopt(socketfd, SOL_SOCKET, SO_SNDBUF, (char *)&
15	SendBufferSize, sizeof(Sen dBufferSize));

7.2.11. ioctlsocket

이 함수는 소케트장치에 특정한 추가선택을 부여하기 위해 사용한다.

```
int ioctlsocket(int sockFd, long cmd, u_long * argp);
```

여기서 두번째 인자는 소케트가 수행할 지령을 나타내며 세번째 인자는 지령에 대한 인자이다. ioctlsocket을 사용하여 핵심부에 요청할수 있는 작업은 다음과 같이 나누어 볼수 있다.

- File operation
- Socket operation
- Routing operation
- Interface operatIon

표 7-6은 요청할수 있는 작업들을 간단히 설명한것이다.

표 7-6. ioctlsocket() 추가선택

구분	요 청	설 명	자료형
File	FIOCLEX	독립적인 사용을 설정	
	FIONCLEX	독립적인 사용을 삭제	
	FIONBIO	비블로크화I/O의 지우기/설정	int
	FIOASYNC	비동기I/O의 지우기/설정	int
	FIONREAD	읽으려는 바이트수의 얻기	int
	FIOSETOWN	owner설정	int
	FIOGETOWN	owner얻기	int
Socket	SIOCSHIWAT	high-water표식설정	int
	SIOCSHIWAT	hlgh_water 표식얻기	int
	SIOCSLOWAT	low_water 표식설정	int
	SIOCGLOWAT	low_water표식얻기	int
	SIOCATMARK	At out_of_band mark	int
	SIOCSPGRP	처리그룹설정	int
	SIOCGPGRP	처리그룹얻기	int
Routing	SIOCADDRT	경로기추가	struct rtentry
	SIOCDELRT	경로기삭제	struct rtentry
intellace	SIOCSIFADDR	ifnet주소설정	struct ifreq
	SIOCGIFADDR	ifnet주소얻기	struct ifreq
	SIOCSIFFLAGS	ifnet기발설정	struct ifreq
	SIOCGIFFLAGS	ifnet기발얻기	struct ifreq
	SIOCGIFCONT	ifnet목록얻기	struct conf
	SIOCSIFDSTADDR	점대점접속(point-to-point) 주소설정	struct ifreq
	SIOCGIFDSTADDR	점대점접속(point-to-point) 주소얻기	struct ifreq
	SIOCGIFBRDADDR	방송추가 얻기	struct ifreq
	SIOCSIFBRDADDR	방송추가 설정	struct ifreq
	SIOCFIGNEMASK	Get net addr mask	struct ifreq
	SIOCSIFNEMASK	Set net addr mask	struct ifreq
	SIOCGIFMETRIC	IF metric얻기	struct ifreq
	SIOCSIFMETRIC	IF metrIc설정	struct ifreq
	SIOCSARP	ARP entry설정	struct arpreq
	SIOCCARP	ARP entry얻기	struct arpreq
	SIOCDARP	ARP entry삭제	struct arpreq

아래의 원천코드는 ioctlsocket를 사용하여 소케트를 비블로크화 (nonblocking)한 실례이다.

원천코드:

```
1  u_long ulNonBlocked = 1;
2  if(ioctlsocket(sockfd, FIONBIO, (u_long * )&u1NonBlockde) < 0)
3  {
4          close(sockfd);
5          printf("ioctlsocket error\r\n");
6          return FALSE;
7  }
```

이렇게 하면 소케트를 생성한후 소케트에 추가선택항목을 추가하여줌으로써 련결을 무작정 기다리는 봉사기나 의뢰기가 조종권을 획득하여 다른 작업을 할수 있게 한다.

7.2.12. 입출력 다중화(Input/Output Multiplexing)

한개의 처리기가 두개이상의 의뢰기로부터 자료를 읽어야 한다고 생각해보자. 처리기는 두개이상의 소케트를 열고 자료가 전송되기를 기다려야 한다. 하지만 의뢰기로부터 자료가 언제 전송되여올지 모르기때문에 한쪽 소케트만 자료를 읽으려고 기다리면 다른쪽 소케트에 자료가 전달되여오더라도 봉사기는 전혀 인식하지 못하는 경우가 있다.

이러한 경우를 극복하기 위한 방도의 하나가 바로 입출력다중화인데 먼저 ioctlsocket를 리용하여 읽으려고하는 두개 이상의 소케트를 비블로크화(Nonblocking)하는 방법이 있다. 그러면 처리기는 차례로 소케트를 읽고 일정한 시간을 송신요구훑기하게 된다. 여기서 매 소케트를 일정한 간격으로 송신요구훑기를 하고 읽어들일 자료가 존재하지 않으면 휴식한다. 하지만 송신요구훑기의 방법은 너무나 많은 체계자원을 소비하는 방법으로서 그리 좋은 방법은 아니다.

또한 처리기가 스레드를 생성하여 각 스레드로 하여금 소케트의 정보를 읽어들이게 하는 방법이 있다. 이때 각 스레드는 소케트에서 자료를 읽어들일 때까지 블로크(block)화 되여도 상관없다. 스레드는 자료를 읽으면 프로쎄스들사이통신(IPC)을 통하여 부모처리기에 읽어들인 자료를 전달한다.

다른 하나의 방법은 select()함수를 리용하는것인데 이것은 처리기가 핵심부에 여러개의 사건(event)을 기다리게 하고 그중 요구하는 사건이 도달되면 처리기를 깨우도록 하는 방법이다. 이렇게 함으로써 처리기는 검색하려고 하는 자료를 읽는것이다. 우의 방법들중 가장 일반적으로 사용되는 방법이 select()함수를 리용하는것이다.

```
int select (int nfds, fd_set * readfds, fd_set *writefds, fd_set
exceptfds, struct timeval * timeout) ;
FD_ZERO(fd_set * set) ;
FD_SET(int fd, fd_set * fdset) ;
FD_CLR(int fd, fd_set * fdset) ;
FD_ISSET(int fd, fd_set * fdset) ;

select()함수에서 사용된 timeval 인자는 다음과 같은 구조체의 지적자이다.

struct timeval {
long tv_sec; /* seconds */
long tv_use; /* and microseconds */
};
```

례를 들어 select()함수를 리용하면 소케트묶음 {1,4,5}가 소케트로부터 자료를 읽어 들일 준비가 되여있고 소케트묶음 {2,7}은 소케트로 자료를 쓸(write) 준비가 되여있으며 소케트묶음 {1,4}는 현재 소케트가 미결(pending)이라는 등의 상태를 알수 있다.

Select()함수는 서술자(descriptor)를 검사한후 그 결과를 반환하는데 이때 timeval 인자는 시간을 나타내는 지적자이고 시간마감(timeout)값은 0이여야 한다. 이러한 작업을 송신요구홅기라고 한다. 그리고 서술자(descriptor)상의 어떤 인자라도 사용할 준비가 되여있으면 지정된 시간을 기다리지 않고 즉시 반환한다. 이때 timeval인자는 검사할 시간을 나타내는 지적자이고 0이외의 값이 설정되여야 한다.

다음은 select()함수의 간단한 사용실례이다.

원천프로그람:

1	struct timeval timeout;
2	timeout.tv_sec = atol(argv[1]);
3	timeout.tv_usec = atol(argv[2]);
4	
5	if(select(0, (fd_set *)0, (fd_set *)0, &timeout) < 0) {
6	printf("select error\r\n");
7	return FALSE;
8	}
9	return TRUE;

우의 실례에서는 select()함수의 두번째, 세번째, 네번째인자를 Null로 설정했지만 만일 검사하려는 파일서술자가 있으면 파라메터(readfds, writefds, exceptfds)안에 값을 설정해야 한다. 이때 네번째 인자인 exceptfds를 통하여 알수 있는 그외의 정보는 다음과 같은 정황에서 사용할수 있다.

• 소케트를 통한 긴급자료의 도착을 알릴 때
• 현재 조종상태정보를 알고 싶을 때

체계에서 사용가능한 최대서술자의 개수는 FD_SETSIZE를 통해 설정이 된다. 례를 들어 다음과 같이 설정이 된 경우에는 최대개수가 64개로 제한이 된다. 그리고 매개의 비트는 서술자와 련결된다.

```
#define FD_SETSIZE      64
Fd_set fdvar
FD_ZERO(&fdvar); //fd_set 를 초기화
FD_SET(1, &fdvar) ; //fd 1bit 검색
FD_SET(4, &fdvar); //fd 4bit 검색
FD_SET(5, &fdvar); //fd 5bit 검색
```

작업을 수행할 때 fd_set()를 초기화하는것은 대단히 중요하다. 만일 초기화하지 않고 사용한다면 체계는 알수 없는 오유들을 발생시킬수 있다. Select()함수를 리용하여 그 결과를 readfds, writfds 그리고 exceptfds에 각각 반환하면 FD_ISSET마크로를 사용하여 fd_set구조체에 select()함수가 정상적으로 값을 반환하였는가를 알아보아야 한다.

FD_ISSET마크로는 select를 통하여 반환된 총합을 반환한다. 만약 시간초과로 반환되였다면 그것은 0이고 오유가 발생하였다면 그 값은 -1이다.

Select()함수는 몇개의 소케트가 동시에 련결되는 프로그람에서 사용되기도 한다.

례를 들면 처리기가 여러개의 소케트를 열고 의뢰기의 련결을 기다리고있다면 하나의 련결이 수락될 때까지 처리기는 다른 소케트를 검사할수 없지만 select()함수를 사용하여 일정한 시간동안 차례로 소케트를 검사하게 함으로써 모든 소케트가 동시에 련결을 기다리게 할수 있다.

제3절. 소케트프로그람작성

이 절에서는 지금까지 소개한 내용과 체계호출함수에 기초하여 기본적인 소케트프로그람을 작성해 보도록 한다. 이러한 과정을 거쳐야만 함수에 대한 정확한 리해와 사용법을 숙련할수 있으며 앞으로 프로그람작성에서 응용할수 있도록 기초를 다질수 있다.

7.3.1. sockaddr_un 구조체(struct)를 리용한 통신

처음에 작성할 프로그람은 sockaddr_un구조체를 리용하여 소케트를 위한 특수한 파일을 작성하고 이것을 리용하여 봉사기와 의뢰기가 통신하도록 하는 프로그람이다. 그러면 먼저 봉사기를 만들어보자.

sockaddr_un구조체를 위하여 다음과 같이 머리부파일을 불러들이고 소케트통신에서 사용할 통보문의 길이를 선언하도록 한다.

```
#include <sys/un.h>
/* 소케트를 통해 주고받을 통보문의 길이 */
#define MSGLEN 32
```

그리고 봉사기를 위한 구조체와 의뢰기를 위한 구조체를 선언하고 통보문송수신을 위한 완충기를 선언하도록 한다.

```
struct sockaddr_un svrAddr;
struct sockaddr_un cltAddr;
/* 통보문 송수신을 위한 완충기의 선언 */
char readbuf[MSGLEN] , sendbuf[MSGLEN] ;
```

통신에서 사용할 소케트를 생성하기전에 기존의 소케트를 먼저 제거하도록 한다. 그다음 socket()함수의 호출을 통하여 소케트서술자를 얻어온다.

```
/*기존의 소케트를 제거한 다음 소케트 생성 및 소케트 서술자 GET */
unlink( "newSocket" ) ;
svrSock = socket(AF_UNIX, SOCK_STREAM, 0);
```

그리고 봉사기쪽 구조체를 설정하고 bind()함수를 실행한 다음 listen()함수를 실행시키도록 한다.

```
/* sockaddr_un 구조설정*/
svrAddr.sun_family = AF_UNIX;
strcpy(svrAddr.sun_path, "newSocket" ) ;

/* svrSock 와 svrAddr 를 리용하여 bind 수행 */
svrLen_ sizeof(svrAddr) ;
bind(svrSock, (struct sockaddr *)&svrAddr, SvrLen) ;
listen(svrSock, 5) ;
```

마지막으로 의뢰기와의 접속을 수락하고 의뢰기로부터 통보문의 전송을 기다린다. 통보문이 접수되면 접수된 통보문을 가공한 다음 의뢰기에 전송하도록 한다.

```
/* 의뢰기와의 통신, accept() 수행 */
int cltLen = sizeof(cltAddr);
int    cltsock    =    accept(svrSock,    (struct    sockaddr
*)&cltAddr,&cltLen);
read(cltSock, readbuf, MSGLEN);

/* 수신된 통보문을 재전송한후 련결해제 */
sprintf(sendbuf,"<재전송> %s" ,readbuf);
write(cltSock, sendbuf, MSGLEN);
close(cltSock);
```

그러면 이번에는 의뢰기의 작성에 대하여 보도록 하자. 의뢰기에서는 봉사기와 마찬가지로 머리부파일들을 불러들이고 통보문길이를 선언해준다. 그 다음 필요한 변수들을 선언하고 통보문의 송수신을 위한 완충기를 선언한다. 모든 초기작업이 끝났으면 socket()함수의 실행을 통하여 소케트의 서술자를 얻어 온다.

그리고 sockaddr_un구조체를 설정한 다음 소케트서술자와 구조체를 리용하여 봉사기와의 련결을 시도한다.

```
/* 의뢰기에서 사용할 소케트 서술자 */
int sockFd = socket(AF_UNIX, SOCK_STREAM, 0);

/* sockaddr_un 구조체 설정*/
sockAddr.sun_family = AF_UNIX;
strcpy(sockAddr.sun_path, "newSocket");

/* 봉사기의 소케트와 련결을 시도 */
int addrLen = sizeof(sockAddr);
connect(sockFd, (struct sockaddr *)&sockAddr, addrLen);
```

마지막으로 봉사기에 통보문을 전송한 다음 봉사기에서 보내오는 통보문을 기다린다. 봉사기로부터 통보문을 받으면 이것을 화면에 출력하고 소케트를 닫는다.

```
/* 봉사기로 통보문을 전송한후, 통보문을 수신*/
write(sockFd, msgbuf, MSGLEN) ;
read(sockFd, msgbuf, MSGLEN) ;

/* 봉사기로부터 수신한 통보문을 출력한후 소케트를 닫음 */
close(sockFd);
```

그러면 지금까지 소개한 내용을 기초로 하여 작성한 전체 원천코드를 보도록 하자. 아래의 내용은 봉사기쪽 코드인 svrUn.c의 원천코드이다.

실례프로그람: svrUn.c	
1	#include <stdio.h>
2	#include <unistd.h>
3	#include <sys/types.h>
4	
5	/* 소케트사용을 위한 머리부파일들 */
6	#include <sys/socket.h>
7	#include <sys/un.h>
8	

```
9   /* 소케트를 통해 주고받을 통보문의 길이 */
10  #define MSGLEN 32
11  
12  int main()
13  {
14          /* 통신을 위한 변수들 선언 */
15          int svrSock, cltSock;
16          int svrLen, cltLen;
17          struct sockaddr_un svrAddr;
18          struct sockaddr_un cltAddr;
19  
20          /* 통보문 송수신을 위한 완충기 선언 */
21          char readbuf[MSGLEN], sendbuf[MSGLEN] ;
22  
23          /* 기존의 소케트를 제거한후 소케트의 생성 및 소케트서술자를 얻기 */
24          unlink("newSocket");
25          svrSock = socket(AF_UNIX, SOCK_STREAM, 0);
26  
27          /* sockaddr_un구조체 설정 */
28          svrAddr.sun_family = AF_UNIX;
29          strcpy(svrAddr.sun_path, "newSocket");
30  
31          /* svrSock과 svrAddr을 리용하여 bind 수행 */
32          svrLen = sizeof(svrAddr);
33          if(bind(svrSock, (struct sockaddr *)&svrAddr, svrLen) < 0)
34          {
35                  fprintf(stderr, "Binding 실패!\n");
36                  exit (1);
37          }
38  
39          /* 통보문을 수신한 다음 전송하는 무한순환고리 */
40          while(1)
41          {
42                  /* 의뢰기와의 통신, Accept()수행 */
43                  cltLen = sizeof(cltAddr);
44                  cltSock = accept(svrSock, (struct sockaddr
    *)&cltAddr,&cltLen);
```

```
45              if(cltSock < 0)
46              {
47                      fprintf(stderr, "Accept()수행 실패!\n");
48                      exit(1);
49              }
50
51              /* 의뢰기로부터 통보문 수신 */
52              printf("의뢰기로부터 통보문 수신 대기중...\n");
53              read(cltSock, readbuf, MSGLEN);
54              printf("수신된 통보문: %s\n", readbuf);
55
56              /* 수신된 통보문을 재전송한후 련결해제 */
57              sprintf(sendbuf, "<재전송> %s",readbuf);
58              write(cltSock, sendbuf, MSGLEN);
59              close(cltSock);
60      }
61 }
```

이번에는 의뢰기쪽 원천코드를 보도록 하자. 아래의것은 의뢰기프로그람인 cltUn.c 파일의 원천코드이다.

실례프로그람: cltUn.c

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4
5  /* 소케트사용을 위한 머리부파일들 */
6  #include <sys/socket.h>
7  #include <sys/un.h>
8
9  /* 소케트를 통해 주고받을 통보문의 길이 */
10 #define MSGLEN 32
11
12 int main()
13 {
14         /* 필요한 변수선언 */
15         Int sockFd, addrLen;
16         struct sockaddr_un sockAddr;
```

17	char msgbuf[MSGLEN];;
18	
19	/* 의뢰기에서 사용할 소케트 서술자 */
20	sockFd = socket(AF_UNIX, SOCK_STREAM, 0);
21	
22	/* sockaddr_un구조체 설정 */
23	sockAddr.sun_family = AF_UNIX;
24	strcpy(sockAddr.sun_path, "newSocket");
25	
26	/* 봉사기의 소케트와 련결을 시도 */
27	addrLen = sizeof(sockAddr);
28	if(connect(sockFd, (struct sockaddr *)&sockAddr, addrLen) == -1)
29	{
30	fprintf(stderr, "봉사기와의 접속에 실패했습니다.!\n");
31	exit (1);
32	}
33	
34	/* 사용자로부터 통보문 입력받음 */
35	printf("전송할 통보문:");
36	gets(msgbuf);
37	
38	/* 봉사기로 통보문전송후, 통보문 수신 */
39	write(sockFd, msgbuf, MSGLEN);
40	read(sockFd, msgbuf, MSGLEN);
41	
42	/* 봉사기로부터 수신한 통보문 출력후 소케트 닫음 */
43	printf("수신한 통보문: %s\n", msgbuf);
44	close(sockFd);
45	exit(0);
46	}

7.3.2. sockaddr_in 구조체를 리용한 통신

이번에는 sockaddr_in구조체를 리용하여 통신프로그람을 작성해보자. 작성할 례제는 앞에서와 마찬가지로 접속대기중인 봉사기에 의뢰기가 접속한 다음 통보문을 전송하고 통보문을 수신한 봉사기는 통보문을 전송한 의뢰기에 답변을 보내는 프로그람으로 하자.

그러면 먼저 봉사기프로그람인 echoSvr프로그람의 작성과정을 살펴보자. 다음과 같이 필요한 머리부파일을 불러들이고 변수를 선언한다.

```
/* 소케트 사용을 위한 머리부파일들 */
#include <sys/socket . h>
#include <netinet/in . h>
#include <arpa/inet . h>

/* 소케트를 통해 주고받을 통보문의 길이 */
#define MSGLEN 32
struct sockaddr_in svrAddr;
struct sockaddr_in cltAddr;
```

socket()함수를 리용하여 소케트를 만들고 봉사기쪽 sockaddr_in구조체를 설정한다. 설정이 끝나면 bind()함수와 listen()함수를 다음과 같이 설정한다.

```
/* sockaddr_in 구조설정*/
svrAddr.sin_family = AF_INET;
svrAddr.sin_addr.s_addr = inet_addr( "127.0.0.1" );
svrAddr. sin_port = 9999;

/* 봉사기를 위한 소케트 생성 및 소케트 서술자 얻기 */
int svrSock = socket(AF_INET, SOCK_STREAM, 0) ;
/* svrSock 과 svrAddr 을 리용하여 bind 수행 */
bind(svrSock, (struct sockaddr *)&svrAddr, sizeof(svrAddr));
/* Listen 체계호출 수행 */
listen(svrSock, 5);
```

이번에는 accept()를 리용하여 의뢰기와의 접속을 허락하고 의뢰기쪽의 소케트 서술자를 얻는다. 그 다음 의뢰기소케트의 서술자를 리용하여 의뢰기에서 보내온 통보문을 읽는다. 마지막으로 읽은 통보문을 의뢰기에 전송한다. 전송이 끝나면 의뢰기소케트를 닫는다.

```
/* 의뢰기와의 통신, Accept()수행 */
int cltLen = sizeof(cltAddr);
int cltSock = accept(svrSock, ( struct sockaddr
*)&cltAddr,&cltLen);

/* 의뢰기로부터 통보문을 수신 */
read(cltSock, readbuf, MSGLEN);

/* 수신된 통보문을 재전송한 후 련결해제*/
sprintf(sendbuf,"<재전송> %s" ,readbuf);
write(cltSock, sendbuf, MSGLEN);
close(cltSock);
```

그러면 지금까지 소개한 내용을 기초로 하여 작성된 echoSvr프로그람의 전체 원천코드를 보도록 하자.

실례프로그람: echoSvr.c	
1	#include <stdio.h>
2	#include <unistd.h>
3	#include <sys/types.h>
4	
5	/* 소케트사용을 위한 머리부파일들 */
6	#include <sys/socket.h>
7	#include <sys/un.h>
8	#include <netinet/in.h>
9	#include <arpa/inet.h>
10	
11	/* 소케트을 통해 주고받을 통보문의 길이 */
12	#define MSGLEN 32
13	
14	int main()
15	{
16	/* 통신을 위한 변수들 선언 */
17	int svrSock, cltSock;
18	int svrLen, cltLen;
19	struct sockaddr_in svrAddr;
20	struct sockaddr_in cltAddr;
21	
22	/* 통보문 송수신을 위한 완충기 선언 */
23	char readbuf[MSGLEN], sendbuf[MSGLEN];
24	
25	/* 봉사기를 위한 소케트 생성 및 완충기 선언 */
26	svrSock = socket(AF_INET, SOCK_STREAM, 0);
27	if(svrSock < 0)
28	{
29	fprintf(stderr, "socket() 실행 실패!\n");
30	exit (1);

```
31          }
32
33          /* sockaddr_in구조체 설정 */
34          svrAddr.sin_family = AF_INET;
35          svrAddr.sin_addr.s_addr = inet_addr ("127.0.0.1");
36          svrAddr.sin_port = 9999;
37
38          /* svrSock과 svrAddr을 리용하여 bind 수행 */
39          svrLen = sizeof(svrAddr) ;
40          if(bind(svrSock, (struct sockaddr *)&svrAddr, svrLen) < 0)
41          {
42                  fprintf(stderr, "Binding 실패!\n" );
43                  exit (1) ;
44          }
45
46          /* Listen체계호출 수행 */
47          if(listen (svrSock, 5) < 0)
48          {
49                  fprintf(stderr, "Listening 실패!\n");
50                  exit (1);
51          }
52
53          /* 통보문을 수신한 다음 전송하는 무한순환고리 */
54          while(1)
55          {
56                  /* 의뢰기와의 통신, Accept()수행 */
57                  cltLen = sizeof(cltAddr);
58                  cltSock = accept(svrSock, (struct sockaddr
    *)&cltAddr,&cltLen);
59                  if(cltSock < 0)
60                  {
61                          fprintf(stderr, "Accept() 수행 실패!\n");
62                          exit(1);
63                  }
```

64	
65	/* 의뢰기로부터 통보문 수신 */
66	printf("의뢰기로부터 통보문 수신 대기중. . .\n");
67	read(cltSock, readbuf, MSGLEN);
68	printf("수신된 통보문: %s\n",readbuf);
69	
70	/* 수신된 통보문을 재전송한후 련결 해제 */
71	sprintf(sendbuf, "<재전송> %s",readbuf);
72	write(cltSock, sendbuf, MSGLEN);
73	close(cltSock);
74	}
75	}

이번에는 echoSvr프로그람과 통신을 수행할 의뢰기프로그람인 echoClt프로그람을 작성하여보자.

먼저 필요한 머리부파일을 include하고 변수들을 선언한다. 그 다음 sockaddr_in 구조체를 다음과 같이 설정한다.

```
/* 접속할 봉사기의 IP 등의 정보를 설정*/
struct sockaddr_in sockAddr;
sockAddr.sin_family = AF_INET;
sockAddr.sin_addr.s_addr = inet_addr( "127.0.0.1" ) ;
sockAddr.sin_port = 9999;
```

이번에는 socket()함수를 리용하여 소케트를 만들고 connect()함수를 리용하여 봉사기의 소케트와 련결한다.

```
/* 의뢰기에서 사용할 소케트 서술자 */
int sockFd = socket(AF_INET, SOCK_STREAM, 0);
/* 봉사기의 소케트와 련결을 시도 */
int result = connect(sockFd, (struct sockaddr*)&sockAddr,
sizeof(sockAddr));
```

봉사기와 접속이 되였으면 사용자로부터 입력받은 통보문을 전송한다. 그리고 봉사기로부터 통보문을 수신받은 다음 소케트를 닫고 프로그람의 실행을 완료한다.

```
/* 사용자로부터 통보문을 입력*/
char msgbuf[MSGLEN];
gets(msgbuf);
/* 봉사기로 통보문전송/통보문의 수신후 소케트를 닫는다 */
write(sockFd, msgbuf, MSGLEN);
read(sockFd, msgbuf, MSGLEN);
close(sockFd);
```

그러면 지금까지 설명한 내용에 기초하여 echoClt프로그람의 전체 원천코드를 보도록 하자.

실례프로그람: echoClt프로그람	
1	#include <stdio.h>
2	#include <unistd.h>
3	#include <sys/types.h>
4	
5	/* 소케트사용을 위한 머리부파일들 */
6	#include <sys/socket.h>
7	#include <netinet/in.h>
8	#include <arpa/inet.h>
9	
10	/* 소케트를 통해 주고받을 통보문의 길이 */
11	#define MSGLEN 32
12	
13	int main()
14	{
15	/* 필요한 변수선언 */
16	int sockFd, addrLen, result;
17	struct sockaddr_in sockAddr;
18	char msgbuf[MSGLEN];;
19	
20	/* 의뢰기에 사용할 소케트 서술자 */
21	sockFd = socket(AF_INET, SOCK_STREAM, 0);
22	
23	/* 접속할 봉사기의 IP 등 정보 설정 */
24	sockAddr.sin_family = AF_INET;
25	sockAddr.sin_addr.s_addr = inet_addr ("127.0.0.1");
26	sockAddr.sin_port = 9999;

```
27
28          /* 봉사기의 소케트과 련결 시도 */
29          addrLen = sizeof(sockAddr);
30          result = connect(sockFd, (struct sockaddr *)&sockAddr, addrLen);
31          if (result == -1)
32          {
33                  fprintf(stderr, "봉사기와의 접속에 실패했습니다. !\n");
34                  exit (1);
35          }
36
37          /* 사용자로부터 통보문 입력받기 */
38          printf("전송할 통보문:");
39          gets(msgbuf);
40
41          /* 봉사기로 통보문 전송후, 통보문 수신 */
42          write(sockFd, msgbuf, MSGLEN);
43          read(sockFd, msgbuf, MSGLEN);
44
45          /* 봉사기로부터 수신한 통보문 출력후 소케트 닫음 */
46          printf("수신한 통보문: %s\n", msgbuf);
47          close(sockFd);
48          exit(0);
49  }
```

우의 프로그람을 콤파일하고 두개의 말단(또는 두개의 쉘창)을 리용하여 실행시켜보자. 정확히 실행이 되면 그림 7-3, 7-4와 같은 결과가 나타날것이다.

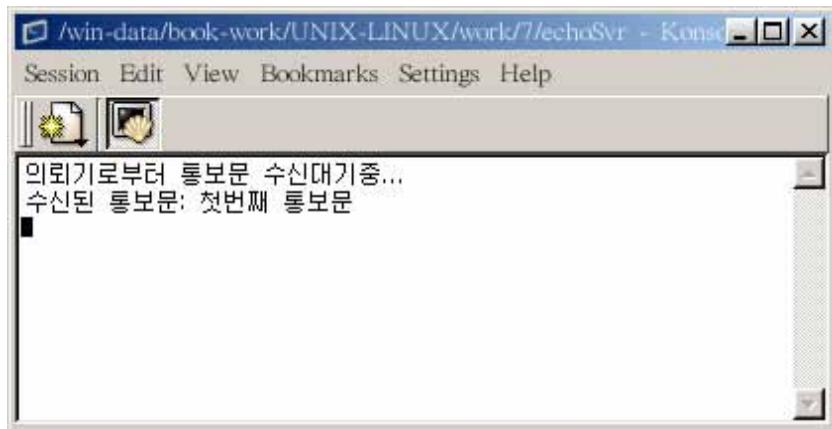



그림 7-3. echoSvr.c 의 실행결과

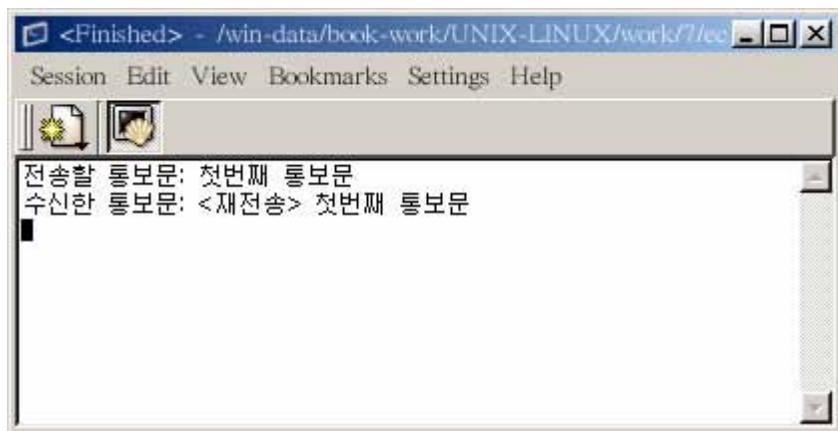



그림 7-4. echoClt.c 의 실행결과

상식

domain name(령역이름)

인터네트와 같은 대규모망의 콤퓨터나 사용자를 식별하기 위한 이름이다.

나라, 조직의 종류, 조직의 명칭, 조직안에 설치되여 있는 콤퓨터 등을 령역(domain)이라고 부르는 론리적인 묶음들로 나누고 계층적으로 배렬한것이 령역이름이다.

간단히 실례를 들어 본다면 www.abc.com.kp 라는 령역이름은 조선민주주의인민공화국(kp)에 있는 기업소(com)로서 abc 라는 이름을 가진 기업소에 있는 www 라는 콤퓨터를 가리킨다.

제8장

체계간 통신 (2)

서론

이 장에서는 7장에서 체계호출함수를 리용하여 소케트프로그람을 작성했던것과는 달리 그것을 좀 더 심화시킨 내용을 취급하게 된다.

그러나 알아두어야 할것은 여기에서 취급하는 소케트프로그람실례들이 실지 프로그람작성에서 제기되는 문제들 가운데서 극히 일부분에 지나지 않는다는것이다. 물론 여기서 다루는 내용은 대부분 프로그람작성의 기초이므로 먼저 이 내용들을 완전히 자기의것으로 소화해야 한다. 그 다음 자기 스스로 계속 응용해보고 문제점들을 바로잡아 나가야 한다.

이 장에서는 TCP와 UDP봉사를 리용한 주콤퓨터의 정보수집 실례와 초과시간을 적용한 통보문의 송수신 실례를 소개한다. 차례를 보면 다음과 같다.

목표

1. 주콤퓨터의 정보수집(UDP, TCP)
2. 통보문의 송수신(초과시간)

제1절. 주콤퓨터의 정보수집(UDP, TCP)

이 절에서 취급하는 내용은 Linux에서 제공하는 봉사를 리용하여 해당 주콤퓨터의 정보를 수집하고 현시하는것이다. 이때 사용할 봉사는 주콤퓨터의 날자와 시간을 제공하는 날자 및 시간(daytime)봉사로서 13번포구를 리용하게 된다.

Linux에서 제공하는 봉사를 확인하기 위해 쓰는 파일은 /etc/services인데 파일의 내부를 보면 봉사이름과 포구이름 등이 기입되여 있다. 이때 daytime과 관련된 내용은 다음과 같다.

```
# /etc/services
daytime 13/tcp
daytime 13/udp
```

8.1.1. UDP 를 리용한 실례

먼저 udp 13번포구를 리용하여 프로그람을 작성해보자. 봉사프로그람의 작성에 앞서 UDP규약과 관련된 내용을 정리해보면 UDP에서는 소케트의 생성 및 사용에 리용하는 규약으로서 SOCK_DGRAM을 지정하게 된다. 그리고 소케트가 생성 및 사용되면 이것은 특정한 IP를 위해서만 사용되는것이 아니기때문에 다른 임의의 IP로도 파케트를 보낼수 있다. 그리고 UDP는 비련결형봉사이기때문에 련결설정을 위해 connect()함수를 사용하지 않는다.

이제 프로그람의 작성과정을 보자. 먼저 필요한 머리부파일들을 불러들인 다음 사용할 변수들을 선언한다. 그 다음 socket()함수를 리용하여 소케트를 만든다. 이때 다음과 같이 SOCK_DGRAM으로 지정을 한다.

```
int sockFd = socket(AF_INET,SOCK_DGRAM,0);
```

그리고 sockaddr_in구조체를 리용하여 필요한 정보를 설정하면서 포구는 13번을 리용하도록 만든다.

```
struct sockaddr__in sockAddr;
bzero( (char * )&sockAddr, sizeof(sockAddr) );
sockAddr . sin_family = AF_INET;
sockAddr . sin_addr. s_addr = inet_addr(argv[1] );
sockAddr . sin_port = htons (13);
```

설정이 끝났으면 주콤퓨터에 정보를 요청하고 보내온 통보문을 수신한다.

```
char readBuf[128] ;
int readCnt;
sendto(sockFd, readBuf, 128, 0, (struct sockaddr*)&sockAdd
r, sizeof(sockAddr));
readCnt = recvfrom(sockFd, readBuf, 128, 0, NULL, NULL);
```

마지막으로 수신된 통보문을 화면에 출력하고 소케트를 닫는다.

```
readBuf[readCnt] = '\0';
printf("주콤퓨터의 daytime: %s\n", readBuf);
close(socFd);
```

그러면 지금까지 설명한 내용에 기초하여 작성된 전체 원천코드를 보도록 하자.

실례프로그람: testUDP.c	
1	#include <stdio.h>
2	#include <stdlib.h>
3	#include <string.h>
4	
5	/* 소케트사용을 위한 머리부파일들 */
6	#include <sys/socket.h>
7	#include <netinet/in.h>
8	#include <arpa/inet.h>
9	
10	int main (int argc, char *argv[])
11	{
12	/* 통신을 위한 변수선언 */
13	int sockFd;
14	struct sockaddr_in sockAddr;
15	
16	/* 자료읽기를 위한 변수선언 */
17	char readBuf[128];
18	int readCnt;
19	
20	/* 호스트명이 입력되였는지 검사 */
21	if(argc != 2)
22	{
23	fprintf(stderr, "\nUsage: testUDP hostname\n\n");
24	exit(1);
25	}
26	
27	/* SOCK_DGRAM을 리용하여 소케트 생성 */
28	sockFd = socket(AF_INET,SOCK_DGRAM, 0);
29	if(sockFd < 0)

30	{
31	fprintf(stderr, "socket open error\n");
32	exit(1);
33	}
34	
35	/* 봉사기와 포구정보를 리용하여 sockAddr 설정 */
36	bzero((char *)&sockAddr, sizeof(sockAddr));
37	sockAddr.sin_family = AF_INET;
38	sockAddr.sin_addr.s_addr = inet_addr(argv[1]);
39	sockAddr.sin_port = htons(13);
40	
41	/* 봉사기로부터 통보문 수신 */
42	sendto(sockFd,readBuf, 128, 0, (struct sockaddr *)&sockAddr, sizeof(sockAddr));
43	readCnt = recvFrom(sockFd, readBuf, 128, 0, NULL, NULL);
44	
45	/* 봉사기로부터 얻어온 daytime정보 출력 */
46	readBuf[readCnt] = '\0';
47	printf("호스트의 daytime: %s\n", readBuf);
48	close(sockFd);
49	exit(0);
50	}

8.1.2. TCP를 리용한 실례

이번에는 TCP를 리용하여 주콤퓨터의 정보를 얻어오는 과정을 살펴보자. TCP는 련결형규약으로서 필요한 접속을 하기 위해서는 매번 소케트를 만들어야 한다. 그러므로 UDP보다 체계자원을 많이 사용한다는 약점이 있다. 그러나 오유발생시 파케트의 재전송이나 중복된 파케트를 제거하는 등 안정한 통신을 지원할수 있다.

이번의 실례에서는 앞에서와 마찬가지로 주콤퓨터의 daytime봉사를 리용하여 정보를 얻어오는 과정을 수행하게 된다. 하지만 사용되는 포구는 TCP포구 13번을 리용하게 되고 정보를 얻어오는 과정이 약간 다르다. 즉 gethostbyname()함수를 리용하여 내부에 설정된 주콤퓨터의 주소를 얻고 getservbyname()함수를 리용하여 봉사되는 포구번호를 얻게 된다.

gethostbyname()함수를 리용하여 주콤퓨터의 주소를 얻을 때 사용되는 파일은 /etc/hosts로 다음과 같은 정보를 리용하게 된다.

```
# /etc/hosts
192.168.8.100 jshin loghost
192.168.8.101 jshin shin
```

그러면 프로그람을 작성하는 과정에 대하여 보도록 하자. 먼저 필요한 머리부파일을 불러들이고 변수들을 선언하도록 한다. 이때 이전과는 달리 netdb.h파일도 포함시킨다. 그 다음 gethostbyname()을 리용하여 다음과 같이 주콤퓨터정보를 얻는다. 주콤퓨터정보를 얻었으면 우선 주콤퓨터와 관련된 정보들을 화면에 현시한다.

```
#include <netdb.h>
struct hostent *host = (struct hostent*) gethostbyname (argv[1]);

char *hostname = hostT->h_name; /* hostname 출력... */
char **alias = hostT->h_aliases; /* alias 출력... */
char **addrList = hostT->h_addr_list; /* addrList 출력... */
```

주콤퓨터정보를 제대로 얻어왔으면 이번에는 daytime봉사의 포구번호를 얻어 오도록 한다. 다음과 같이 servent구조체와 getservbyname()함수를 리용하도록 한다.

```
/* "/etc/services" 파일안의 daytime 봉사포구정보 */
struct servent *servT = (struct servent*) getservbyname
("daytime", "tcp");
```

이제 소케트를 만들고 앞에서 얻어온 주소와 포구정보를 리용하여 봉사기와 접속을 진행하도록 한다.

```
struct sockaddr_in sockAddr;
int sockFd = socket (AF_INET, SOCK_STREAM, 0);
sockAddr. sin_family = AF_INET;
sockAddr. sin_port = servT->s_port;
sockAddr. sin_addr = * (struct in_addr *) *hostT->h_addr_list;
connect (sockFd, (struct sockaddr *)&sockAddr, sizeof
(sockAddr));
```

봉사기와 접속이 되였으면 봉사기로부터 자료를 수집 및 출력한 다음 소케트를 닫는다.

```
/* 자료읽기를 위한 변수선언 */
char readBuf[128];
int readCnt = read(sockFd; readBuf, sizeof(readBuf));
readBuf[readCnt] = '\0';
printf("현재 시간: %s\n\n\n" , readBuf);
close(sockFd);
```

그러면 지금까지 설명한 내용에 기초하여 작성된 프로그람의 전체 원천코드를 보도록 하자.

실례프로그람: hostinfo.c

```
1   #include <stdio.h>
2   #include <unistd.h>
3   #include <sys/types.h>
4   
5   /* 소케트 및 호스트 정보를 위한 머리부파일들 */
6   #include <netdb.h>
7   #include <sys/socket.h>
8   #include <netinet/in.h>
9   #include <arpa/inet.h>
10  
11  int main (int argc, char *argv[])
12  {
13          /* 통신을 위한 변수선언 */
14          int sockFd, addrLen;
15          struct sockaddr_in sockAddr;
16  
17          /* 호스트정보를 위한 변수선언 */
18          char **alias, **addrList;
19          struct hostent *hostT;
20          struct servent *servT;
21  
22          /* 자료읽기를 위한 변수선언 */
23          char readBuf[128];
24          int readCnt;
25  
26          /* 인수가 있으면 해당 호스트이름 사용, 아니면 localhost 사용 */
27          if(argc == 2)
28                  hostT = (struct hostent *)gethostbyname(argv[1]);
29          else
30                  hostT = (struct hostent *)gethostbyname("localhost");
31          if(hostT == NULL)
32          {
33                  fprintf(stderr, "호스트정보를 얻어올수 없습니다. \n");
34                  exit(1);
35          }
```

36	
37	/*/etc/hosts 속의 해당 호스트별명 현시 */
38	printf("\n\n<<호스트:%s>> \n", hostT->h_name);
39	alias = hostT->h_aliases;
40	readCnt = -1;
41	while(*alias)
42	{
43	printf("호스트별명[%d]: %s\n", readCnt, *alias);
44	alias++;
45	readCnt++;
46	}
47	
48	/*/etc/hosts속의 해당 호스트주소 현시 */
49	addrList = hostT->h_addr_list;
50	readCnt = 1;
51	while(*addrList)
52	{
53	printf("IP주소[%d]: %s\n",
54	readCnt,
55	inet_ntoa(*(struct in_addr *) * addrList));
56	addrList++;
57	readCnt++;
58	}
59	/*/etc/services 파일안의 daytime봉사 포구정보 */
60	servT = (struct servent*)getservbyname("daytime", "tcp");
61	if(servT = NULL)
62	{
63	fprintf(stderr, "daytime봉사를 리용할수 없습니다. \n");
64	exit(1);
65	}
66	printf("DATAIME봉사 포구번호<TCP>: %d\n", ntohs(servT->s_port));
67	
68	/* 소케트의 생성 및 봉사기의 정보 설정 */
69	sockFd = socket(AF_INET, SOCK_STREAM, 0);
70	sockAddr.sin_family = AF_INET;
71	sockAddr.sin_port = servT->s_port;
72	sockAddr.sin_addr = * (struct in_addr *) *hostT->h_addr_list;

```
73         addrLen = sizeof(sockAddr) ;
74
75         /* 봉사기의 daytime 포구에 접속을 시도 */
76         if(connect(sockFd, (struct sockaddr *)&sockAddr, addrLen) < 0)
77         {
78                 fprintf(stderr, "connection에 실패했습니다.. \n");
79                 exit (1);
80         }
81
82         /* daytime봉사를 통해 호스트의 현재 시간을 현시 */
83         readCnt = read(sockFd, readBuf, sizeof(readBuf));
84         readBuf[readCnt] = '\0';
85         printf("현재시간: %s \n\n\n", readBuf);
86         close(sockFd);
87         exit(0);
88 }
```

프로그람에 대한 분석 및 작성이 모두 끝나면 실행시켜보자.
실행시키면 그림 8-1과 같은 결과가 나온다.

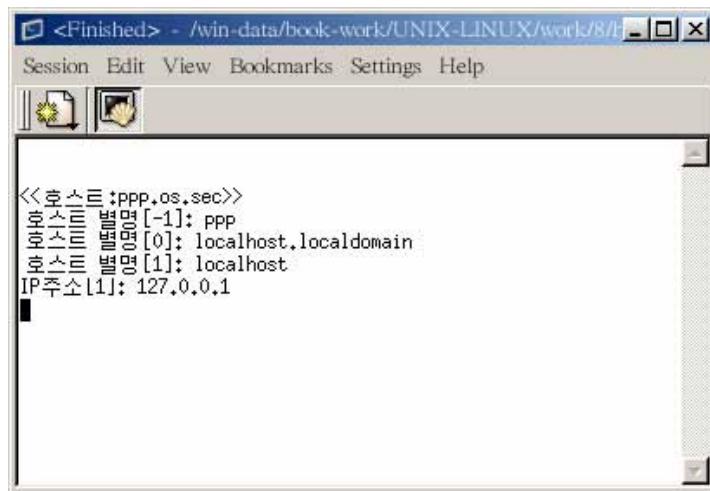



그림 8-1. hostInfo.c 의 실행결과

무선망이란 어떤것인가?

무선망이란 말 그대로 전자기파나 빛(적외선 포함)등 통신선이외의 전송경로를 리용한 근거리통신망(LAN)을 말한다. 무선망은 신호의 전송매질로 대기권을 사용한 망을 념두에 두기도 한다.

무선망은 그의 구축이 빠르고 쉽다는 우점은 있으나 아직은 그를 실현하는데 많은 비용이 든다는 약점도 가지고있다.

제2절. 통보문송수신(시간초과)

이 절에서는 7장에서 다루었던 echoSvr와 echoClt프로그람을 변형시켜 봉사기와 의뢰기가 통보문을 계속 주고받도록 만들어보자. 그리고 이 과정에 봉사기가 응답을 늦게 하면 의뢰기에서 《시간초과(Time out)》를 알리는 통보문이 출력되도록 하는 기능을 추가하자.

시간초과기능을 추가하기 위해서는 박자계수기를 설정하고 검사하여야 하는데 이것은 7장에서 설명한 select()함수를 리용하여 할수 있다. 그리고 FD_SET()와 timeval구조체를 리용하여 소케트와 시간을 설정하며 select가 소케트대면부를 검사하도록 만든다.

먼저 박자계수기를 위한 변수를 선언하고 시간초과를 위한 시간을 설정한다.

```
struct timeval tv;
tv.tv_sec = 10;
tv.tv_usec = 0;
```

그 다음 소케트를 만들고 FD_SET를 리용하여 비트를 설정한다.

```
int sockFd = socket(AF_INET, SOCK_STREAM, 0);
fd_set rset;
FD_ZERO(&rset);
FD SET(sockFd, &rset);
```

이제 select()를 리용하여 박자계수기의 실행 및 소케트의 검사를 해보도록 하자.

```
if(select(sockFd+1, &rset, NULL, NULL, &tv) == 0)
{
      fprintf(stderr,". . .TIME OUT. . .\n");
      exit(1);
}
```

그러면 이러한 시간초과과정을 적용하여 의뢰기프로그람인 sockClt를 작성해보자. sockclt는 봉사기와 접속을 한 다음에는 무한순환을 돌면서 통보문을 주고받게 된다. 그러다가 시간초과가 발생하거나 《quit》라는 문자렬을 입력받게 되면 무한순환에서 탈퇴하게 된다. 실지 과제에서도 소케트통신도중에 시간초과가 발생하면 자료전송을 중지하고 련결을 새롭게 맺는 과정이 필요하게 된다.

sockClt 프로그람의 원천코드는 다음과 같다.

실례프로그람: sockClt.c	
1	#include <stdio.h>
2	#include <unistd.h>
3	#include <sys/types.h>
4	
5	/* 소케트사용을 위한 머리부파일들 */
6	#include <sys/socket.h>
7	#include <netinet/in.h>
8	#include <arpa/inet.h>
9	
10	/* 소케트를 통해 주고받을 통보문의 길이 */
11	#define MSGLEN 32
12	
13	int main()
14	{
15	/* 자료송수신을 위한 변수선언 */
16	int sockFd, addrLen, result;
17	struct sockaddr_in sockAddr;
18	char readbuf[MSGLEN], sendbuf[MSGLEN];
19	
20	/* 시간초과를 위한 변수선언 */
21	int retSelect;
22	struct timeval tv;
23	fd_set rset;
24	
25	/* 초과시간을 10s로 설정 */
26	tv.tv_sec = 10;
27	tv.tv_usec = 0;
28	

29	/* 의뢰기에서 사용할 소케트서술자 */
30	sockFd = socket(AF_INET, SOCK_STREAM, 0);
31	if(sockFd < 0)
32	{
33	fprintf(stderr, "socket open error\n");
34	exit(1);
35	}
36	
37	/* 접속할 봉사기의 IP 등 정보 설정 */
38	sockAddr.sin_family = AF_INET;
39	sockAddr.sin_addr.s_addr = inet_addr ("127.0.0.1");
40	sockAddr.sin_port = 9999;
41	
42	/* 봉사기의 소케트와 련결을 시도 */
43	addrLen = sizeof(sockAddr);
44	if (connect(sockFd, (struct sockaddr *)&sockAddr, addrLen) == -1)
45	{
46	fprintf (stderr, "봉사기와의 접속에 실패했습니다. !\n");
47	exit (1);
48	}
49	
50	/* 통보문 송수신을 위한 무한순환고리 */
51	while(1)
52	{
53	
54	/* 입력받은 통보문전송, "quit"이면 무한순환고리에서 탈출 */
55	printf("전송할 통보문:");
56	gets(sendbuf);
57	write(sockFd, sendbuf, MSGLEN);
58	if(!strncmp(sendbuf, "quit" ,4)) break;
59	
60	/* 시간초과 기동, select()를 리용하여 sockFd 검사 */
61	FD_ZERO(&rset);
62	FD_SET(sockFd, &rset);
63	retSelect = select(sockFd+1, &rset, NULL, NULL, &tv);
64	if(retSelect == 0)
65	{

66	fprintf(stderr, "...TIME OUT...\ n");
67	break;
68	}
69	
70	/* 봉사기로부터 수신한 통보문출력 */
71	read(sockFd, readbuf, MSGLEN);
72	printf("수신한 통보문: %s\n", readbuf);
73	}
74	close(sockFd);
75	exit(0);
76	}

이번에는 봉사기측 프로그람인 sockSvr프로그람을 작성해보자. 먼저 필요한 머리부 파일들을 불러들이고 변수들을 선언하도록 한다. 그 다음 소케트를 만들고 bind를 실행한다.

원천코드:	
1	/* 소케트를 통해 주고받을 통보문의 길이 */
2	#define MSGLEN 32
3	int svrSock = socket(AF_INET, SOCK_STREAM, 0) ;
4	
5	/* sockaddr_in구조체의 설정 */
6	svrAddr.sin_family = AF_INET ;
7	svrAddr.sin_addr.s_addr = htonl(INADDR_ANY) ;
8	svrAddr.sin_port = 9999;
9	
10	/* svrSock과 svrAddr을 리용하여 bind 수행 */
11	bind(svrSock, (struct sockaddr *)&svrAddr, sizeof (svrAddr)) ;
12	
13	/* Listening… 수행 */
14	listen(svrSock, 5) ;

이제 의뢰기와 접속을 수락한 다음 무한순환을 돌면서 의뢰기와 통보문을 주고받도록 한다. 만일 의뢰기가 보내온 통보문에 'quit'가 포함되면 실행을 중지하고 소케트를 닫게 된다.

원천코드:	
1	/* 의뢰기와의 통신 Accept 수행 */
2	int cltLen = sizeof (cltAddr) ;
3	int cltSock = accept (svrSock, (struct sockaddr *)&cltAddr, &cltLen) ;

4	
5	/* 무한순환고리를 돌면서 통보문 접수 및 송신 */
6	char readbuf[MSGLEN], sendbuf[MSGLEN];
7	read(cltSock, readbuf, MSGLEN);
8	printf("수신된 통보문: %s\n", readbuf);
9	if(!strncmp(readbuf, "quit", 4))
10	break;
11	write(cltSock, sendbuf, MSGLEN);
12	
13	/* 무한순환고리를 탈출한후 소케트를 닫기 */
14	close(cltSock);
15	close(svrSock);

그러면 이번에는 sockSvr프로그람의 전체 원천코드를 보도록 하자.

실례프로그람: sockSvr.c	
1	#include <stdio.h>
2	#include <unistd.h>
3	#include <sys/types.h>
4	
5	/* 소케트사용을 위한 머리부파일들 */
6	#include <sys/socket.h>
7	#include <netinet/in.h>
8	#include <arpa/inet.h>
9	
10	/* 소케트을 통해 주고받을 통보문의 길이 */
11	#define MSGLEN 32
12	
13	int main()
14	{
15	/* 통신을 위한 변수들 선언 */
16	int svrSock, cltSock;
17	int svrLen, cltLen;
18	struct sockaddr_in svrAddr;
19	struct sockaddr_in cltAddr;
20	
21	/* 통보문 송수신을 위한 완충기선언 */
22	char readbuf[MSGLEN], sendbuf[MSGLEN];
23	

24	/* 봉사기를 위한 소케트의 생성 및 소케트서술자 얻기*/
25	svrSock = socket(AF_INET, SOCK_STREAM, 0);
26	if(svrSock < 0)
27	{
28	fprintf(stderr, "socket () 실행 실패!\n");
29	exit (1);
30	}
31	
32	/* sockaddr_in구조체 설정 */
33	svrAddr.sin_family = AF_INET;
34	svrAddr.sin_addr.s_addr = htonl("INADDR_ANY");
35	svrAddr.sin_port = 9999;
36	
37	/* svrSock과 svrAddr을 리용하여 bind()수행 */
38	svrLen = sizeof(svrAddr);
39	if(bind(svrSock, (struct sockaddr *)&svrAddr, svrLen) < 0)
40	{
41	fprintf(stderr, "Binding 실패!\n");
42	exit (1);
43	}
44	
45	/* Listening … 수행 */
46	if(listen (svrSock, 5) < 0)
47	{
48	fprintf(stderr, "Listening 실패!\n");
49	exit (1);
50	}
51	
52	/* 의뢰기와의 통신, Accept()수행 */
53	cltLen = sizeof (cltAddr);
54	cltSock = accept(svrSock, (struct sockaddr *)&cltAddr,&cltLen);
55	if(cltSock < 0)
56	{
57	fprintf(stderr, "Accept() 수행 실패!\n");
58	exit(1);
59	}

```
60
61          while (1)
62          {
63                  /* 통보문 수신, "quit"을 수신하면 무한순환고리를 탈출 */
64                  printf("의뢰기로부터 통보문 수신 대기중. . .\n");
65                  read(cltSock, readbuf, MSGLEN);
66                  printf("수신된 통보문: %s\n", readbuf);
67                  if(!strncmp (readbuf, "quit",4)) break;
68
69                  /* 처리지연으로 인한 시간초과 시험! */
70                  /* sleep (20); */
71
72                  /* 수신된 통보문을 재전송 */
73                  sprintf(sendbuf, "<재전송> %s",readbuf);
74                  write(cltSock, sendbuf, MSGLEN);
75          }
76
77          /* 봉사기와 의뢰기의 소케트를 모두 닫고 완료 */
78          close(cltSock);
79          close(svrSock);
80          exit (0);
81  }
```

프로그람작성이 끝나면 콤파일을 진행하고 다음과 같이 두개의 쉘창이나 원격으로 떨어진 두개의 체계를 리용하여 봉사기와 의뢰기를 각각 실행시키도록 한다.

결과는 그림 8-2와 같다.

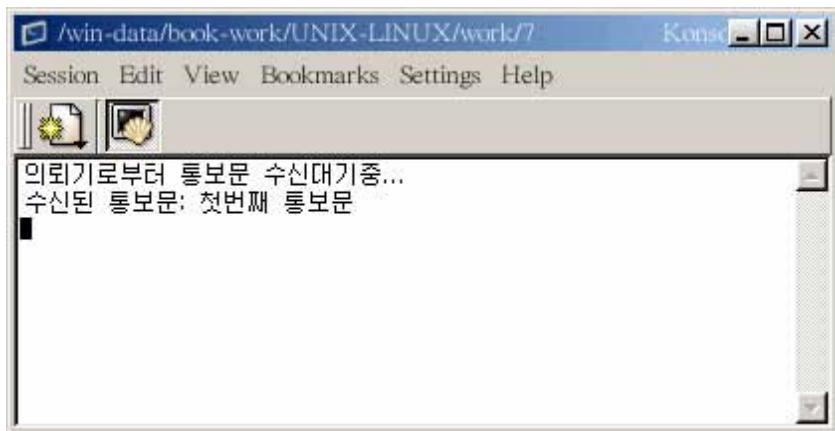



그림 8-2. sockSvr.c의 실행결과

만일 10s이내에 봉사기로부터 응답이 없으면 그림 8-3, 8-4와 같은 결과가 나온다.

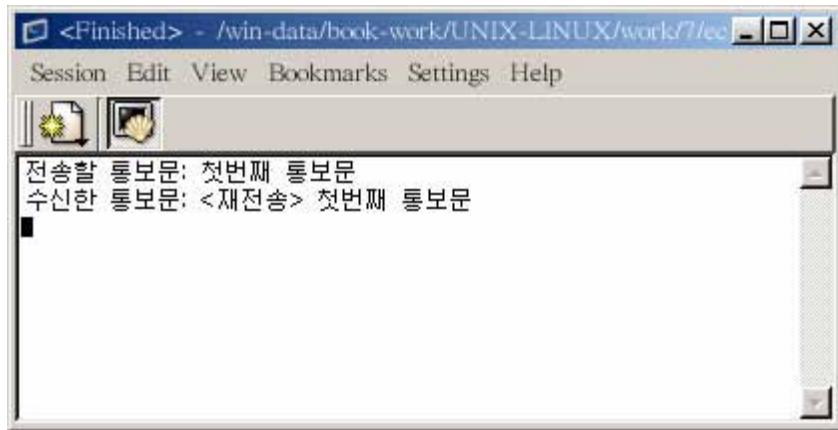



그림 8-3. sockClt.c의 실행결과(1)

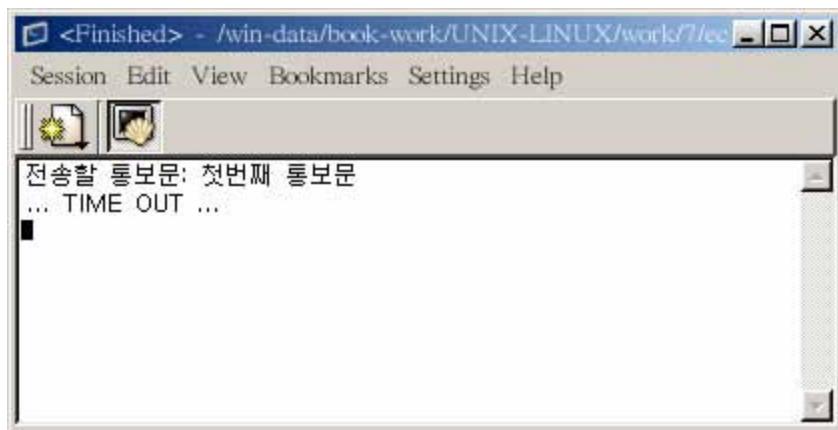



그림 8-4. sockClt.c의 실행결과(2)

찾아보기

Linux망프로그람작성법

집 필	박종혁	심 사	최광철
편 집	림일남	교 정	박석채
장 정	서경애	콤퓨터편성	여은정

낸 곳	교육성 교육정보쎈터	인쇄소	교육성 교육정보쎈터
인 쇄	주체 97(2008)년 8월 10일	발 행	주체 97(2008)년 8월 20일

교-07-1306